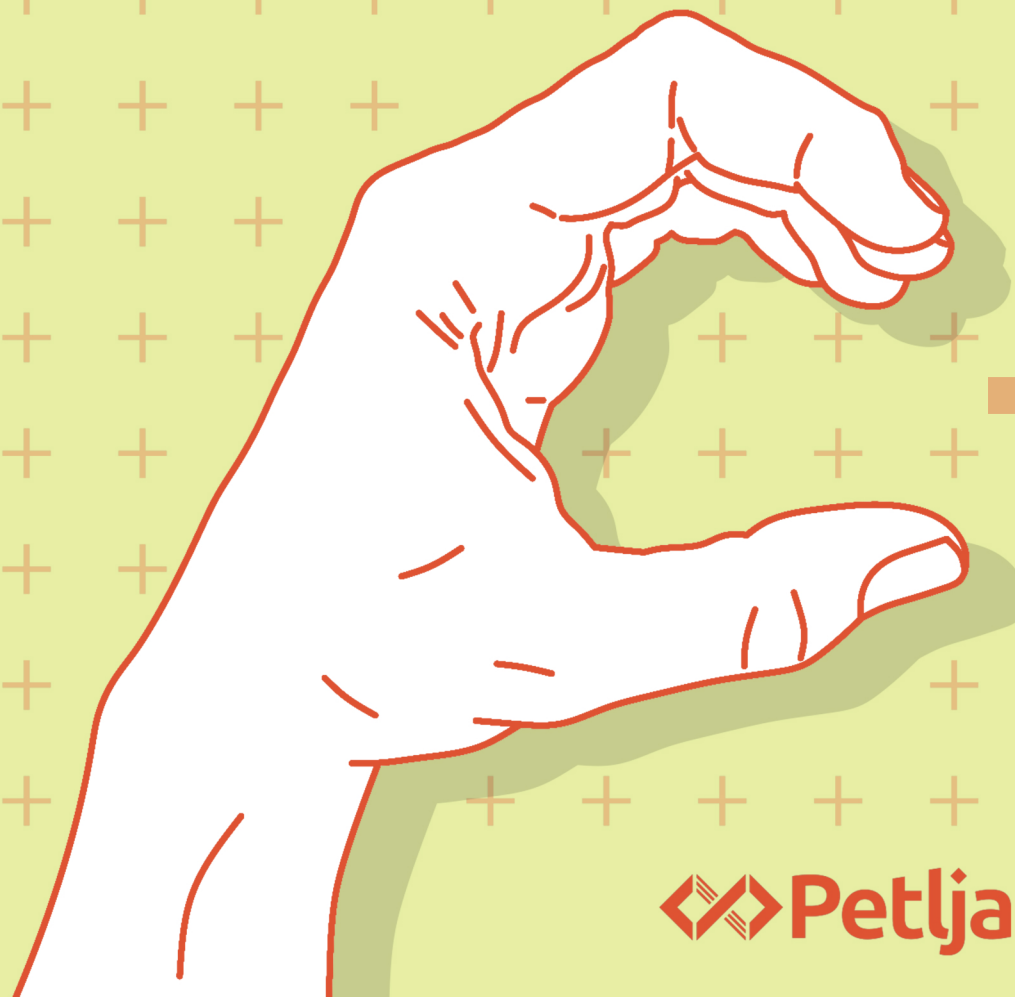
Методичка збирка задатака из алгоритмике - елементарне технике анализе и конструкције алгоритама - C++



Petlja

Друштво математичара Србије
Фондација Петља

Методичка збирка задатака из алгоритмике — C++

елементарне технике конструкције и анализе ефикасних алгоритама

Београд, 2019.

Аутори:

Филип Марић, професор на Математичком факултету у Београду
Милан Вуџелија, фондација „Петља”
Весна Маринковић, доцент на Математичком факултету у Београду
Станка Матковић, професор у Математичкој гимназији у Београду
Душко Обрадовић, професор у гимназији „Вељко Петровић” у Сомбору
Милан Чабаркапа, професор у Математичкој гимназији у Београду

Методичка збирка задатака из алгоритмике — C++
елементарне технике конструкције и анализе ефикасних алгоритама

Издавачи: Друштво математичара Србије и Фондација „Петља”
За издвача: *др Војислав Андрић*

Обрада текста и илустрације: *аутори*, Дизајн корица: *Иван Авдић*

Издање: 1.1 (електронско)

Садржај

1 Анализа коректности алгоритама **1**
1.1 Облици испитивања коректности . . . 2
1.2 Неке честе грешке у програмима . . . 2
1.3 Индуктивно-рекурзивна конструкција . . . 3
1.3.1 Доказ коректности рекурзивних функција . . . 4
1.3.2 Доказ коректности итеративних алгоритама - инваријанте петље . . . 5
Задатак: Аритметички троугао . . . 8
Задатак: Тробојка . . . 10
Задатак: Двобојка . . . 13
Задатак: Први који није дељив . . . 13
Задатак: Најмањи број који није збир елемената скупа . . . 17
Задатак: Бинарни запис . . . 19
Задатак: Растављање на просте чиниоце . . . 19

2 Сложеност израчунавања **22**
2.1 Мерење времена извршавања . . . 23
2.2 Асимптотска анализа сложености . . . 25
2.3 Сложеност неких честих облика петљи . . . 28
2.4 Скривена сложеност . . . 31
2.5 Савети за побољшање сложености . . . 31
2.6 Замена итерације формулом . . . 32
2.6.1 Аритметички и геометријски низ . . . 32
2.6.2 Збирови степена . . . 32
2.6.3 Комбинаторика . . . 32
Задатак: Спортске припреме . . . 33
Задатак: N-ти дан тренинга . . . 35
Задатак: Аритметички троугао . . . 35
Задатак: Аритметички квадрат . . . 38
Задатак: Број подстрингова који почињу и завршавају са 1 . . . 38
Задатак: Недостајући број . . . 40
Задатак: Број дељивих у интервалу . . . 41
Задатак: Хлебови . . . 43
Задатак: Дељиви око броја . . . 43
Задатак: Максимални принос . . . 43
Задатак: Растављања на збир узастопних . . . 45
Задатак: Пријатељски бројеви . . . 47
Задатак: Највећи заједнички делилац . . . 47
2.7 Одсецање . . . 51
Задатак: Прост број . . . 51
Задатак: Ератостеново сито . . . 54
Задатак: Збир простих прост . . . 58
Задатак: Серија сјајних партија . . . 60
Задатак: Најдужа серија победа . . . 62
Задатак: Број растућих сегмената . . . 63
Задатак: Максимални збир сегмента . . . 63
Задатак: Својство 132 . . . 66

2.8 Сортирање . . . 68
2.8.1 Функције сортирања . . . 68
Задатак: Сортирање бројева . . . 69
Задатак: Медијана . . . 71
Задатак: Цик цак селекција . . . 73
2.8.2 Сортирање по разним критеријумима . . . 75
Задатак: Сортирање такмичара . . . 76
Задатак: Сортирање на основи растојања од O . . . 81
Задатак: Највреднији предмети . . . 83
Задатак: Сортирање линија . . . 84
Задатак: Сортирање по просеку . . . 86
Задатак: Ранг сваког елемента . . . 90
2.8.3 Сортирање разврставањем . . . 90
Задатак: Сортирање пребројавањем . . . 91
Задатак: Разврставање по општинама . . . 93
Задатак: Разврставање по првом слову . . . 97
Задатак: Сортирање бројева вишеструким разврставањем (RadixSort) . . . 97
2.8.4 Обрада дупликата (поновљених вредности у низу) . . . 99
Задатак: Дупликати . . . 100
Задатак: Највећи поновљени елемент . . . 102
Задатак: Двоструки студент . . . 104
Задатак: Неупарени елемент . . . 104
Задатак: Број различитих дужина дужи . . . 105
Задатак: Најбројнији елемент . . . 105
2.8.5 Груписање блиских вредности . . . 105
Задатак: Праведна подела чоколадица . . . 105
Задатак: Најближе собе . . . 107
Задатак: Најбројнији подскуп који садржи узастопне целе бројеве . . . 107
2.8.6 Свођење на канонски облик . . . 107
Задатак: Анаграм . . . 108
Задатак: Провера пермутација . . . 109
Задатак: D-пермутација . . . 109
Задатак: Анаграми . . . 112
2.8.7 Сортирање интервала . . . 114
Задатак: Најбројнији пресек интервала . . . 114
Задатак: Покривање праве затвореним интервалима . . . 117
2.8.8 Остале примене сортирања . . . 118
Задатак: Хиршов h-индекс . . . 118
Задатак: Збир минимума тројки . . . 118
Задатак: Коректни телефони . . . 118
2.9 Бинарна претрага . . . 119
2.9.1 Бинарна претрага елемента у низу . . . 120
Задатак: Провера бар-кодова . . . 120
Задатак: Број парова датог збира . . . 124
Задатак: Квадрати . . . 125
Задатак: Ранг сваког елемента . . . 128
Задатак: i-ти на месту i . . . 128
Задатак: Штерн-Брокоово стабло . . . 130
2.9.2 Бинарна претрага преломне тачке . . . 132
Задатак: Први већи и последњи мањи . . . 132
Задатак: Провера бар-кодова . . . 139
Задатак: Број такмичара изнад прага . . . 141
Задатак: Тастатура и миш . . . 141
Задатак: Кружне зоне . . . 143
Задатак: Разлика висина . . . 144
Задатак: Најближи датом елементу . . . 148
Задатак: Оптимални сервис . . . 148
Задатак: Први који није дељив . . . 149

Задатак: К-најближих датом . . . 153
Задатак: Врх планине . . . 159
Задатак: i-ти на месту i . . . 159
Задатак: Минимум ротираног сортираног низа . . . 159
Задатак: Претрага ротираног сортираног низа . . . 160
2.9.3 Проналажење оптималне вредности решења бинарном претрагом . . . 164
Задатак: Дрва . . . 164
Задатак: Конференција . . . 165
Задатак: Најкраћа подниска која садржи све дате карактере . . . 169
Задатак: Хиршов h-индекс . . . 173
Задатак: Највећи квадрат у хистограму . . . 173
Задатак: Муцајући подниз . . . 173
Задатак: Пуно фигурица . . . 174
Задатак: Кувар . . . 174
Задатак: Градња . . . 175
Задатак: Гласници . . . 175
2.10 Скупови и мапе (речници) . . . 178
2.10.1 Скупови . . . 178
2.10.2 Мапе (речници) . . . 179
Задатак: Дупликати . . . 179
Задатак: Неупарени елемент . . . 181
Задатак: Двоструки студент . . . 182
Задатак: Најбројнији елемент . . . 182
Задатак: Сортирање бројева . . . 183
Задатак: Квадрати . . . 183
Задатак: Највећи поновљени елемент . . . 183
Задатак: Број различитих дужина дужи . . . 183
Задатак: Број парова датог збира . . . 183
Задатак: Тројке датог збира (3sum) . . . 184
Задатак: Анаграм . . . 184
Задатак: Провера пермутација . . . 185
Задатак: D-пермутација . . . 185
Задатак: Анаграми . . . 186
2.11 Инкременталност . . . 188
2.11.1 Инкременталност збира и производа . . . 188
Задатак: Префикс највећег збира . . . 188
Задатак: Сегмент датог збира у низу целих бројева . . . 191
Задатак: Факторијели од 1 до n . . . 193
Задатак: Хармонијски пи . . . 195
Задатак: Сума реда . . . 195
Задатак: Оптимални сервис . . . 195
Задатак: Најкраћи сегмент збира бар K . . . 195
2.11.2 Инкременталност минимума и максимума . . . 197
Задатак: Најбољи “сабмит” . . . 197
Задатак: Поглед на реку . . . 199
Задатак: Ред особа . . . 200
Задатак: Најдужи сегмент који садржи узастопне бројеве . . . 200
Задатак: Добри такмичари . . . 203
2.11.3 Инкременталност - остале статистике . . . 206
Задатак: Рутер . . . 206
Задатак: Централа . . . 209
Задатак: Највећи тежински збир после цикличног померања . . . 211
Задатак: Дрва . . . 216
Задатак: Хиршов h-индекс . . . 218
Задатак: Максимални збир сегмента фиксираног почетка . . . 219
2.11.4 Покретни прозор . . . 220
Задатак: Пермутоване подниске . . . 220
Задатак: Сегмент највећег просека . . . 226

Задатак: Панграми . . . 226
2.11.5 Инкременталност - вишедимензионални низови . . . 227
Задатак: Суме трапеза . . . 227
Задатак: Минимуми правоугаоника . . . 231
2.11.6 Инкременталност - суфикси до сваке позиције у низу . . . 231
Задатак: Број растућих сегмената . . . 231
Задатак: Најдужа серија победа . . . 232
Задатак: Серија сјајних партија . . . 232
Задатак: Број сегмената исте парности . . . 232
Задатак: Број сегмената са различитим елементима . . . 233
Задатак: Максимални збир сегмента . . . 235
2.12 Збирови префикса и разлике суседних елемената низа . . . 237
Задатак: Аритметички троугао . . . 237
Задатак: Збирови сегмената . . . 238
Задатак: Максимални збир сегмента . . . 238
Задатак: Најкраћи сегмент збира бар K . . . 239
Задатак: Сегмент датог збира у низу целих бројева . . . 241
Задатак: Сегмент датог збира у низу природних бројева . . . 241
Задатак: Сегменти чији је збир дељив са k . . . 241
Задатак: Сегмент највећег збира дељивог дужином низа . . . 244
Задатак: Први сегмент дељив са N . . . 244
Задатак: Слаткиши за сав новац . . . 244
Задатак: Кружни пут . . . 245
Задатак: Збирови правоугаоника . . . 248
Задатак: Пар производа у ранцу . . . 251
Задатак: Попуњавање празнина . . . 254
Задатак: Збир подниза карата . . . 255
Задатак: Највећи НЗД низа након замене једног елемента . . . 255
Задатак: Увећавање сегмената . . . 258
Задатак: Најбројнији пресек интервала . . . 260
Задатак: Пермутација са највећим збиром упита . . . 260
2.13 Техника два показивача . . . 261
Задатак: Обједињавање . . . 261
Задатак: Сортирани квадрати бројева . . . 266
Задатак: Заједнички елементи три уређена низа . . . 268
Задатак: Близанци . . . 268
Задатак: Прости чиниоци 235 . . . 268
Задатак: Тастатура и миш . . . 270
Задатак: Двобојка . . . 270
Задатак: Тробојка . . . 278
Задатак: Оптимални сервис . . . 282
Задатак: Број парова датог збира . . . 282
Задатак: Тројке датог збира (3sum) . . . 282
Задатак: Разлика висина . . . 282
Задатак: Сегмент датог збира у низу природних бројева . . . 288
Задатак: Пуно фигурица . . . 288
Задатак: Пар производа у ранцу . . . 288
Задатак: Број сегмената са различитим елементима . . . 290
Задатак: Конференција . . . 294
Задатак: Најкраћа подниска која садржи све дате карактере . . . 294
Задатак: Квадратно-троугаони бројеви . . . 294
Задатак: Кружни пут . . . 294
Задатак: Двоструко сортирана претрага . . . 297

3 Конструкција алгоритама рекурзијом тј. индукцијом 303
3.1 Елементарне петље изражене рекурзивно . . . 303
Задатак: Бројеви од a до b . . . 303
Задатак: Бројање у игри жмурке . . . 304

Задатак: Троцифрени парни бројеви . . . 305
Задатак: Одбројавање уназад . . . 305
Задатак: Подела интервала на једнаке делове . . . 306
Задатак: Геометријска серија . . . 307
Задатак: Збир n бројева . . . 307
Задатак: Читање до нуле . . . 307
Задатак: Просек свих бројева до краја улаза . . . 308
Задатак: Прерачунавање миља у километре . . . 308
Задатак: Табелирање функције . . . 308
Задатак: Факторијел . . . 309
Задатак: Степен . . . 309
Задатак: Једнакост растојања . . . 309
Задатак: Средине . . . 309
Задатак: Најнижа температура . . . 310
Задатак: Други на ранг листи . . . 310
Задатак: Разлика сума до мах и од мах . . . 311
Задатак: Негативан број . . . 313
Задатак: Прва негативна температура . . . 313
Задатак: Последња негативна температура . . . 313
Задатак: Бројеви дељиви са 3 . . . 314
Задатак: Просек одличних . . . 315
Задатак: Број и збир цифара броја . . . 315
Задатак: Провера бар-кодова . . . 316
Задатак: Обједињавање . . . 317
Задатак: Број парова датог збира . . . 318
Задатак: Последњих k цифара степена . . . 318
3.2 Репна рекурзија . . . 323
Задатак: Збир n бројева . . . 323
Задатак: Број и збир цифара броја . . . 324
Задатак: Факторијел . . . 325
Задатак: Степен . . . 325
Задатак: Једнакост растојања . . . 326
Задатак: Најнижа температура . . . 327
Задатак: Други на ранг листи . . . 327
Задатак: Прва негативна температура . . . 328
Задатак: Последња негативна температура . . . 329
Задатак: Просек одличних . . . 330
3.3 Извођење итеративних алгоритама из рекурзивних . . . 330
Задатак: Грејов код . . . 330
Задатак: Абацаба . . . 332
Задатак: Морзеов низ . . . 334
Задатак: Избацивање цифара на све начине . . . 337
Задатак: Не садрже цифру 3 . . . 340
Задатак: Дужина сажете форме ниске . . . 343
Задатак: Слово у сажетој форми ниске . . . 347
Задатак: Ханојске куле . . . 350
Задатак: Функција пар-непар . . . 351
3.4 Претрага у дубину . . . 352
Задатак: Број белих области . . . 352
Задатак: Највећа заражена област . . . 355
Задатак: Minesweeper отварање . . . 355
Задатак: P-Q коњи . . . 356
3.5 Рекурзивни спуст . . . 356
Задатак: Превођење потпуно заграђеног израза у постфиксни облик . . . 356
Задатак: Израз у коме нема заграда . . . 358
Задатак: Вредност израза . . . 364
Задатак: Вредност потпуно заграђеног мин-макс израза . . . 366
Задатак: Вредност мин-макс израза . . . 366

4 Генерисање комбинаторних објеката **367**
Задатак: Следећа варијација . . . 367
Задатак: Све варијације . . . 368
Задатак: Све речи од датих слова . . . 370
Задатак: Сви подскупови . . . 371
Задатак: Следећи подскуп . . . 376
Задатак: Сви подскупови лексикографски . . . 379
Задатак: Следећи бинарни низ без суседних јединица . . . 380
Задатак: Сви бинарни низови без суседних јединица . . . 381
Задатак: Бројеви који у бинарном запису немају две суседне нуле . . . 383
Задатак: Следећа комбинација . . . 385
Задатак: Све комбинације . . . 386
Задатак: Све комбинације са понављањем . . . 391
Задатак: Следећа пермутација . . . 392
Задатак: Све пермутације . . . 394
Задатак: Сви палиндроми од дате речи . . . 397
Задатак: Следећа партиција . . . 402
Задатак: Све партиције . . . 404
Задатак: Све једноцифрене партиције . . . 407
Задатак: Сви n-тоцифрени бројеви са датим збиром цифара . . . 408
Задатак: K-та тешка реч . . . 410
Задатак: Сви распореди заграда . . . 414
Задатак: Варијације без понављања . . . 417
Задатак: Разлика суседних цифара k . . . 420
Задатак: Комплетан Грејов код . . . 421

5 Бектрекинг и груба сила **424**
Задатак: Пут кроз лавиринт . . . 424
Задатак: Максимални безбедни пут . . . 428
Задатак: Обојени лавиринт . . . 430
Задатак: Падајућа лоптица . . . 431
Задатак: Распоређивање n дама на шаховској табли . . . 433
Задатак: Судоку . . . 436
Задатак: Латински квадрати . . . 438
Задатак: Скакачева тура . . . 440
Задатак: Сви збирови обиласка матрице . . . 441
Задатак: Број поднизова датог збира . . . 442
Задатак: Мерење са n тегова . . . 447
Задатак: Мерење са n врста тегова . . . 451
Задатак: Број израза дате вредности . . . 454
Задатак: Дужина најдужег проходног пута . . . 458
Задатак: Најдужи пут низбрдо . . . 460
Задатак: Уклањање погрешних заграда . . . 462
Задатак: Братска подела . . . 463
Задатак: Збир суседних пун квадрат . . . 467
Задатак: Подела на палиндромске подниске . . . 468

6 Подели па владај **471**
Задатак: Сортирање бројева . . . 471
Задатак: Медијана . . . 473
Задатак: Збир k најбољих . . . 477
Задатак: Силуета града . . . 482
Задатак: Број инверзија . . . 485
Задатак: Број елемената десно од датог елемента већих или једнаких од њега . . . 488
Задатак: Број сортираних тројки . . . 494
Задатак: Бинарне слике . . . 502
Задатак: Максимални збир сегмента . . . 504
Задатак: Паметна куповина акција . . . 506
Задатак: Најближи пар тачака . . . 506

Задатак: Тримини . . . 512
Задатак: Хоризонт . . . 513
Задатак: Трећи обилазак . . . 513

7 Динамичко програмирање **516**
7.1 Бројање комбинаторних објеката . . . 516
Задатак: Број комбинација . . . 516
Задатак: Број комбинација са понављањем . . . 521
Задатак: Број партиција . . . 524
Задатак: Број цик-цак партиција . . . 530
Задатак: Разлагање на збир квадрата . . . 535
Задатак: Број начина разлагања на збир различитих n-тих степена . . . 538
Задатак: Дигитални бројач . . . 541
Задатак: Низови 0 и 1 без две суседне 1 - редни број . . . 547
Задатак: Број начина декодирања . . . 549
Задатак: Примитивни бинарни записи . . . 553
Задатак: Број појављивања подниске . . . 556
7.2 Оптимизација коришћењем динамичког програмирања . . . 558
Задатак: Едит растојање . . . 558
Задатак: Најдужи заједнички подниз две ниске . . . 562
Задатак: Најдужи подниз који се понавља . . . 565
Задатак: Најдужа заједничка подниска . . . 567
Задатак: Најдужа подниска који се понавља без преклапања . . . 567
Задатак: Најдужи растући подниз . . . 568
Задатак: Падајуће лоптице . . . 574
Задатак: Допуна нулама до највећег скаларног производа . . . 574
Задатак: Исплата са најмање новчића . . . 574
Задатак: Најдужи подниз палиндром . . . 580
Задатак: Максимални збир сегмента . . . 585
Задатак: Оптимално множење матрица . . . 587
Задатак: Максимални збир на путу кроз матрицу . . . 587
Задатак: Максимални пут кроз матрицу . . . 587
Задатак: Цилиндрична матрица . . . 591
Задатак: Подскуп максималног збира дељивог са k . . . 592
Задатак: Превоз предмета лифтом . . . 598

8 Грамзиви алгоритми **605**
Задатак: Реч у реч прецртавањем слова . . . 605
Задатак: Жаба на камењу . . . 608
Задатак: Шаховске екипе . . . 608
Задатак: Ментори . . . 614
Задатак: Распоред активности . . . 614
Задатак: Зли учитељ . . . 618
Задатак: Мали поштар . . . 619
Задатак: Минимална сума два броја формирана од датих цифара . . . 622
Задатак: Најмањи број надовезивањем више бројева . . . 623
Задатак: Излог са два реда књига . . . 624
Задатак: Неопадајућа растојања суседних елемената . . . 625
Задатак: Гласници . . . 626
Задатак: Мост . . . 627

9 Структуре података **631**
9.1 Стек . . . 631
Задатак: Линије у обратном редоследу . . . 631
Задатак: Прегледање веба . . . 632
Задатак: Сажимање путања . . . 632
Задатак: Упареност заграда . . . 632
Задатак: Push-pop реконструкција . . . 633
Задатак: Сажимање серија узастопних једнаких елемената . . . 633

Задатак: Сортирање бројева . . . 633
Задатак: Број белих области . . . 634
Задатак: Minesweeper отварање . . . 635
Задатак: Највећа заражена област . . . 635
Задатак: P-Q коњи . . . 636
Задатак: Вредност постфиксног израза . . . 636
Задатак: Превођење потпуно заграђеног израза у постфиксни облик . . . 636
Задатак: Вредност израза . . . 637
Задатак: Вредност мин-макс израза . . . 640
Задатак: Вредност потпуно заграђеног мин-макс израза . . . 640
Задатак: Фреквенцијски стек . . . 640
Задатак: Стек и максимуми . . . 643
9.2 Ред . . . 643
Задатак: Сегмент највећег просека . . . 643
Задатак: Последњих k линија . . . 644
Задатак: Сортирање - сви испред мањи или сви испред већи . . . 645
Задатак: Јосифов проблем . . . 646
Задатак: Прости чиниоци 235 . . . 647
Задатак: Најдужи сегмент који садржи узастопне бројеве . . . 648
Задатак: Максимална бијекција . . . 648
9.3 Ред са приоритетом . . . 651
Задатак: Сортирање бројева . . . 651
Задатак: Збир k најбољих . . . 652
Задатак: Збирови квадрата . . . 652
Задатак: Степени природних бројева . . . 656
Задатак: K-ти највећи збир пара елемената два низа . . . 659
Задатак: Ажурирање медијане . . . 662
Задатак: Сумњиве трансакције . . . 668
Задатак: Силуета града . . . 669
Задатак: Најкраћи сегмент збира бар K . . . 672
Задатак: Својство 132 . . . 674
9.4 Монотони стек и ред . . . 678
Задатак: Најближи већи претходник . . . 678
Задатак: Најближи већи следбеник . . . 682
Задатак: Највећи правоугаоник у хистограму . . . 682
Задатак: Највећи квадрат у хистограму . . . 682
Задатак: Збир минимума . . . 682
Задатак: Сегменти оивичени максимумима . . . 683
Задатак: Својство 132 . . . 683
Задатак: Max сегмената дужине k . . . 685
Задатак: Најкраћи сегмент збира бар K . . . 691

Глава 1

Анализа коректности алгоритама

Делови текста у наставку су преузети из скрипте "Програмирање 2", аутора Предрага Јаничића и Филипа Марића.

Исправност тј. коректност је суштинска особина алгоритама и програма. Иако се некада у пракси користе програми за које се зна да понекад могу да дају и нетачне резултате, то најчешће није случај и од програма се захтева да буде практично апсолутно непогрешив.

Једно од централних питања у развоју програма је питање његове исправности (коректности). Софтвер је у данашњем свету присутан на сваком кораку: софтвер контролише много тога — од банковних рачуна и компоненти телевизора и аутомобила, до нуклеарних електрана, авиона и свемирских летелица. У свом том софтверу неминовно су присутне и грешке. Грешка у функционисању даљинског управљача за телевизор може бити тек узнемирујућа, али грешка у функционисању нуклеарне електране може имати разорне последице. Најопасније грешке су оне које могу да доведу до великих трошкова, или још горе, до губитка људских живота. Неке од катастрофа које су општепознате су експлозија ракете Ариане 1996. узрокована конверзијом броја из шездесетчетворобитног реалног у шеснаестобитни целобројни запис која је довела до прекорачења, затим пад сателита Криосат (енгл. Cryosat) 2005. године услед грешке у софтверу због које није на време дошло до раздвајања сателита и ракете која га је носила коштао је Европску Унију око 135 милиона евра, затим грешка у нумеричком копроцесору процесора Pentium 1994. узрокована погрешним индексима у петљи `for` у оквиру софтвера који је радио дизајн чипа, као и пад орбитера послатог на Марс 1999. узрокован чињеницом да је део софтвера користио метричке, а део софтвера енглеске јединице. Међутим, фаталне софтверске грешке и даље се непрестано јављају и оне коштају светску економију милијарде долара. Ево неких од најзанимљивијих:

- Не нарочито опасан, али веома занимљив пример грешке је грешка у програму Microsoft Excel 2007 који, због грешке у алгоритму форматирања бројева пре приказивања, резултат израчунавања израза `77.1 * 850` приказује као `100 000` (иако је интерно коректно сачуван).
- У Лос Анђелесу је 14. септембра 2004. године више од четиристо авиона у близини аеродрома истовремено изгубило везу са контролом лета. На срећу, захваљујући резервној опреми унутар самих авиона, до несреће ипак није дошло. Узрок губитка везе била је грешка прекорачења у бројачу милисекунди у оквиру система за комуникацију са авионима. Да иронија буде већа, ова грешка је била откривена раније, али пошто је до открића дошло када је већ систем био испоручен и инсталиран на неколико аеродрома, његова једноставна поправка и замена није била могућа. Уместо тога, препоручено је да се систем ресетује сваких 30 дана како до прекорачења не би дошло. Процедура није испоштована и грешка се јавила после тачно 2^{32} милисекунди, односно 49,7 дана од укључивања система.
- Више од пет процената пензионера и прималаца социјалне помоћи у Немачкој је привремено остало без свог новца када је 2005. године уведен нови рачунарски систем. Грешка је настала због тога што је систем, који је захтевао десетоцифрени запис свих бројева рачуна, код старијих рачуна који су имали осам или девет цифара бројеве допуњавао нулама, али са десне уместо са леве стране како је требало.
- Једна бака у Америци је на свој 106. рођендан добила позив да мора да крене у школу, јер је систем бележио године помоћу две цифре.
- Компаније Dell и Apple морале су током 2006. године да корисницима замене више од пет милиона лап-

топ рачунара због грешке у дизајну батерије компаније Sony која је узроковала да се неколико рачунара запали.

1.1 Облици испитивања коректности

У развијању техника верификације програма, потребно је најпре прецизно формулисати појам коректности тј. исправности програма. Исправност програма почива на појму **спецификације**. Спецификација је, неформално, опис жељеног понашања програма који треба написати. Спецификација се обично задаје у терминима **предуслова** тј. услова које улазни параметри програма задовољавају, као и **постуслова** тј. услова које резултати израчунавања морају да задовоље. Када је позната спецификација, потребно је верификовати програм, тј. доказати да он задовољава спецификацију.

Коректност се огледа кроз два аспекта:

парцијална коректност: свака вредност коју алгоритам израчуна за улазне параметре који задовољавају спецификацију (тј. предуслов) мора да задовољи спецификацију (тј. постуслов).

заустављање алгоритам мора да се заустави за све улазе који задовољавају спецификацију (тј. предуслов).

Већина алгоритама које ћемо проучавати у овом курсу биће потпуни тј. заустављаће се за све допуштене улазе. За заустављајуће парцијално коректне алгоритме кажемо да су **тотално коректни**. Интересантно, доказано је да не постоје алгоритми којима би се испитивала горе наведена својства алгоритама.

Поступак показивања да је програм исправан назива се **верификовање програма**. Два основна приступа верификацији су:

динамичка верификација подразумева проверу исправности у фази извршавања програма, најчешће путем тестирања;

статичка верификација подразумева анализу изворног кода програма, често коришћењем формалних метода и математичког апарата.

Систематично тестирање је сигурно најзначајнији облик постизања високог степена исправности програма. Тестирањем на већем броју исправних улаза и упоређивањем добијених и очекиваних резултата може се открити велики број грешака. Нагласимо и да се тестирањем не може показати да је програм коректан, већ само да није коректан. Наиме, практично никада није могуће испитати понашање програма баш на свим исправним улазима. Већ програм који сабира два 32-битна броја има $2^{32} \cdot 2^{32} = 2^{64}$ исправних комбинација улазних параметара и исцрпно тестирање оваквог програма би трајало годинама. Зато се исцрпно тестирање скоро никада не спроводи, већ се програми тестирају тако да се улази бирају пажљиво, тако да покрију различите гране током извршавања програма. Обично се додатно посебно тестира понашање програма на неким граничним улазима (енгл. edge cases, corner cases), јер програми понекада не обрађују све специјалне случајеве како би требало. Многи системи за учење програмирања (такозвани грејдери, енгл. grader) оцењују ученичка решења тестирањем на већем броју тест-примера и савет почетницима је да током учења обавезно користе овакве системе.

О методама статичке верификације биће више речи у наставку овог поглавља.

1.2 Неке честе грешке у програмима

Сваки исправан програм мора да буде заснован на исправном алгоритму. Дакле, од неисправног алгоритма није могуће направити исправан програм и основна ствар приликом писања исправних програма је да се обезбеди исправност алгоритма који се примењује. Са друге стране, алгоритми се описују често на апстрактнијем нивоу него што су сами програми, и многи детаљи се занемарују. Зато се услед детаља имплементације од исправног алгоритма може добити неисправан програм. Наведимо неке честе грешке.

- Једна од најчешћих грешака представља **грешка прекорачења** (енгл. overflow). Наиме, ако се у имплементацији одабере бројевни тип података којим се не могу исправно представити сви подаци, тада програм даје неисправне резултате. Чест је случај да програмер одабере тип података који може да репрезентује исправно и улазне и излазне вредности, међутим, дешава се да међурезултати не могу да се репрезентују исправно, што се теже примети, а доводи до грешке.

- Честа грешка је **прекорачење граница низа** (енгл. buffer overflow). На пример, ако смо у низу одвојили место за 30 бројева, онда је могуће уписивати вредонсти само на позиције $0, 1, \ldots, 29$. Нарочито је критична позиција 30 (тј. у општем случају позиција n за низ од n елемената). Пошто у савременим програмским језицима бројање позиција у низовима креће од нуле, на позицију n није могуће уписивати вредности. У језику C++ се не врши провера опсега пре приступа елементима низа (тј. вектора) и одговорност је програмера да обезбеди да се не приступа ван граница - у супротном је понашање програма недефинисано, што значи да програм може да настави да ради неисправно и после одређеног броја инструкција да буде прекинут од стране оперативног система, али и да грешка може да прође неопажено. Уколико у петљи у низ уписујемо податке чији број не знамо унапред, неопходно је да пре сваког уписа проверимо да ли се упис врши унутар граница низа (или да користимо неки облик низа који допушта аутоматско проширивање додавањем нових елемената).
- Честа грешка је необраћање пажње на **специјалне случајеве**. На пример, ако у низу тражимо елемент који задовољава неки услов, неопходно је да обезбедимо да програм коректно ради и у случају када ниједан елемент не задовољава тај услов. Треба пажљиво прецизирати да ли функција тада треба да врати број елемената низа или, на пример, -1, и треба осигурати да се у коду који позива ову функционалност добро реагује на ситуацију у којој тражени елемент не постоји. Специјални случајеви најчешће настају када неке вредности не постоје (када је неки скуп чије елементе разматрамо празан), затим када су улазне вредности у неком специјалном односу (на пример, да ли геометријски програм исправно ради ако су унете тачке колинеарне) и слично. При том, треба пажљиво прецизирати спецификацију задатка и одредити који специјални случајеви јесу, а који нису допуштени спецификацијом. Поново је потребно обратити пажњу на то да иако улазни параметри понекада не могу бити у неком специјалном односу, то не значи да међурезултати неће бити у том односу, па је онда потребно програме ипак прилагодити да обрате пажњу на све специјалне случајеве.

1.3 Индуктивно-рекурзивна конструкција

Кључна идеја у конструкцији алгоритама је то да је конструкција алгоритама веома тесно повезана са доказивањем теорема математичком индукцијом. **Математичка индукција**, у свом основном облику, је следећи начин доказивања особина природних бројева. Нека је P произвољно својство које се може формулисати за природне бројеве. Тада важи

$$(P(0) \wedge (\forall n)(P(n) \Rightarrow P(n+1))) \Rightarrow (\forall n)(P(n))$$

Дакле, да бисмо доказали да сваки природан број има неко својство P (тј. да бисмо доказали $(\forall n)(P(n))$), довољно је да докажемо да нула има то својство (тј. $P(0)$) и да докажемо да чим неки број има то својство, има га и његов следбеник (тј. да докажемо $(\forall n)(P(n) \Rightarrow P(n+1))$). Прво тврђење се назива **база индукције**, а друго **индуктивни корак**. Приницип математичке индукције је прилично јасан – на основу базе знамо да 0 има својство P, на основу корака да њен следбеник тј. 1 има својство P, на основу корака да његов следбеник тј. 2 има својство P итд. Интуитивно нам је јасно да на овај начин можемо стићи до било ког природног броја, који сигурно мора имати својство P. База се може формулисати и за веће вредности од нуле, али онда само можемо да тврдимо да елементи који су већи или једнаки од базе имају својство P.

Основни приступ конструкције алгоритама је тзв. **индуктивни** тј. **рекурзивни** приступ. Он подразумева да се решење проблема веће димензије проналази тако што умемо да решимо проблем истог облика, али мање димензије и да од решења тог проблема добијемо решење проблема веће димензије. Притом за почетне димензије проблема решење морамо да израчунавамо директно, без даљег свођења на проблеме мање димензије. Ако се приликом свођења димензија проблема увек смањује, конструисани алгоритми ће се увек заустављати.

- Имплементација алгоритма може бити таква да промењиве унутар петље итеративно ажурирају своје вредности кренувши од вредности које представљају решења елементарних проблема, па до крајњих вредности које представљају решења задатог проблема. Пошто је ово прилично слично принципу математичке индукције, кажемо да је алгоритам дефинисан **индуктивно**.
- Имплементација може бити таква да функција која решава полазни проблем сама себе позива да би решила проблем истог облика, али мање димензије (осим у случају елементарних проблема, који се директно решавају) и тада кажемо да је алгоритам дефинсан **рекурзивно**.

У наставку овог поглавља ћемо се бавити итеративно имплементираним, индуктивно конструисаним алгоритмима. Рекурзија је јако важна техника, којом ћемо се посебно бавити (укључујући и питање коректности рекурзивних функција).

Индуктивна конструкција лежи у основни практично свих итеративних алгоритама које смо до сада разматрали. На пример, алгоритам израчунавања збира серије бројева (на пример, збира елемената неког низа) почива на томе да знамо да израчунамо збир празне серије (то је 0) и да ако знамо збир серије од k елемената, тада умемо да израчунамо и збир серије која се добија проширивањем те серије додатним $k+1$-вим елементом (то радимо тако што дотадашњи збир увећамо за тај нови елемент).

```
int zbir = 0;
for (int i = 0; i < a.size(); i++)
    zbir = zbir + a[i];
```

Дакле, и у овом алгоритму имамо индуктивну базу (која одговара иницијализацији промењиве пре уласка у петљу) и индуктивни корак (који одговара телу петље, у ком се ажурира вредност резултујуће промењиве, у овом случају збира). База може одговарати и случају једночланог (а не обавезно празног) низа, али тада не можемо да гарантујемо да ће алгоритам радити исправно у случају празног низа. То одговара варијанти алгоритма у којој збир иницијализујемо на први елемент низа, па га увећавамо редом за један по један елемент од позиције 1 надаље.

Дефинисање алгоритама индуктивно-рекурзивом конструкцијом је у веома тесној вези са доказивањем њихове коректности. Иако постоје формални оквири за доказивање коректности императивних програма (пре свега *Хорова логика*), ми ћемо се бавити искључиво неформалним доказима и веза између логике у којој вршимо доказивање и (императивног) програмског језика у којем се програм изражава биће прилично неформална.

Рецимо и да ћемо приликом доказивања коректности програма обично игнорисати ограничења записа бројева у рачунару и подразумеваћемо да је опсег бројева неограничен и да се реални бројеви записују са максималном прецизношћу. Дакле, нећемо обраћати пажњу на грешке које могу настати услед прекорачења или поткорачења вредности током извођења аритметичких операција (иако реално то често може бити узрок грешака у програмима).

1.3.1 Доказ коректности рекурзивних функција

Проблем: Дефинисати функцију која одређује минимум непразног низа бројева и доказати њену коректност.

Алгоритам се веома једноставно конструише индуктивно-рекурзивном конструкцијом. Димензија проблема у овом примеру је број елемената низа.

База: Ако низ има само један елемент, тада је тај елемент уједно и минимум.

Корак: У супротном, претпоставимо да некако умемо да решимо проблем за мању димензију и на основу тога покушајмо да добијемо решење за цео низ. Дакле, претпоставимо да је дужина низа $n > 1$ и да умемо да нађемо број m који представља минимум првих $n-1$ елемената низа. Минимум целог низа дужине n је мањи од бројева m и преосталог, n-тог елемента низа (ако бројање креће од 0, то је елемент a_{n-1}).

На основу овога можемо дефинисати рекурзивну функцију.

```
#include <iostream>
using namespace std;

int min2(int a, int b) {
    return a < b ? a : b;
}

int min(int a[], int n) {
    if (n == 1)
       return a[0];
    else {
       int m = min(a, n-1);
       return min2(m, a[n-1]);
```

```
    }
}

int main() {
    int a[] = {3, 5, 4, 1, 6, 2, 7};
    int n = sizeof(a) / sizeof(int);
    cout << min(a, n) << endl;
}
```

Рецимо и да смо уместо дефинисања функције `min2` за одређивање минимума два броја могли користити и функцију `std::min` из заглавља `<algorithm>`. Ипак, у овим програмима нећемо користити специфичне могућности језика C++, да бисмо нагласили да технике које у овом поглављу уводимо нису ни по чему специфичне за тај језик.

Коректност претходног алгоритма се може формулисати у облику следеће теореме.

Теорема: За сваки непразан низ a (низ за који је дужина $|a| \geq 1$) и за свако $1 \leq n \leq |a|$ позив `min(a, n)` враћа најмањи међу првих n елемента низа a (са a и n су обележене вредности низа `a` и променљиве `n`, а са $|a|$ дужина низа `a`).

Ту теорему можемо доказати индукцијом.

- Базу индукције представља случај $n = 1$, тј. позив `min(a, 1)`. На основу дефиниције функције `min` резултат је `a[0]` тј. први члан низа a_0 и тада тврђење тривијално важи (јер је он уједно најмањи међу првих 1 елемената низа).
- Као индуктивну хипотезу можемо претпоставити да ако важи $1 \leq n - 1 < |a|$, тада позив `min(a, n-1)` враћа најмањи од првих $n - 1$ елемената низа a. Из те претпоставке потребно је да докажемо да за n које задовољава $1 < n \leq |a|$ позив `min(a, n)` враћа најмањи од првих n елемената низа a (при том је a непразан низ). На основу дефиниције функције `min`, позив `min(a, n)` ће вратити минимум бројева m (који представља резултат позива `min(a, n-1)`) и a_{n-1}. Пошто су услови индуктивне хипотезе задовољени, на основу индуктивне хипотезе знамо да ће m бити најмањи међу првих $n - 1$ елемената низа a. Зато ће минимум броја m и n-тог елемента низа (елемента a_{n-1}) бити најмањи међу првих n елемената низа a.

Примећујемо огромну сличност између рекурзивне конструкције алгоритма и индуктивног доказа његове коректности. Стога слободно можемо да кажемо да су рекурзија и индукција "две стране исте медаље" (индукцију користимо као технику доказивања, а рекурзију као технику дефинисања функција тј. конструкције алгоритама).

Рецимо и да је овај облик коришћења математичке индукције мало нестандардан, јер се не користи директно индукција по природним бројевима, већ се користи индукција по структури рекурзивне функције у којој се, из претпоставке да сваки рекурзивни позив враћа коректан резултат, доказује да функција враћа коректан резултат. Таква теорема индукције се може доказати на основу класичне математичке индукције по броју рекурзивних позива, под претпоставком да се докаже да се рекурзивна функција увек зауставља.

1.3.2 Доказ коректности итеративних алгоритама - инваријанте петље

Један од основних појмова у анализи и разумевању итеративних програма су **инваријанте петљи**. То су логички услови који важе након сваког извршавања наредби у телу петље, а након извршавања целе петље гарантују коректност алгоритма који та петља имплементира. Инваријанте суштински описују значење свих промењивих унутар петље. Илуструјмо појам инваријанте на једном једноставном примеру.

Размотримо следећу, класичну имплементацију алгоритма за одређивање минимума непразног низа бројева.

```
#include <iostream>
#include <algorithm>
using namespace std;

int minNiza(const vector<int>& a) {
    int m = a[0];
    for (int i = 1; i < a.size(); i++)
        m = min(m, a[i]);
    return m;
```

```
}

int main() {
    vector<int> a{3, 5, 4, 1, 6, 2, 7};
    cout << minNiza(a) << endl;
}
```

У сваком кораку петље, део низа чији минимум знамо постаје дужи за по један елемент. Алгоритам креће од префикса низа дужине 1 и поставља променљиву `m` на вредност првог елемента низа a_0. У сваком кораку петље, претпостављамо да променљива m садржи вредност минимума првих i елемената низа, а онда у телу петље обрађени део низа проширујемо додајући $i+1$-ви елемент низа, на позицији i. Минимум проширеног низа се израчунава као минимум минимума првих i елемената низа (чија је вредност смештена у променљивој `m`) и додатног елемента низа a_i. Након извршавања тела петље, део низа чији минимум је познат је проширен на $i+1$ елемент. На крају петље је i једнако дужини низа, па променљива `m` садржи минимум целог низа.

Пре него што пређемо на формални доказ претходог разматрања, скренимо пажњу на то да именоване величине у математици (тачније алгебри) и у програмирању имају различите особине. Наиме, именоване величине у математици (параметри, непознате) означавају једну вредност док у (императивном) програмирању именоване величине имају динамички карактер и мењају своје вредности током извршавања програма по правилима задатим самим програмом. На пример, бројачка променљива `i` у некој петљи може редом имати вредности 1, 2 и 3. Да бисмо направили разлику између променљивих и њихових текућих вредности, користићемо различит фонт - променљиву програма ћемо обележавати са `i`, а њену вредност са i. Ако желимо да разликујемо стару и нову вредност променљиве `i`, користићемо ознаке i и i'. Ако желимо да нагласимо да је променљива редом узимала неку серију вредности, користићемо ознаке i_0 (почетна вредност променљиве `i`), i_1, i_2, ... У ситуацијама у којима се вредност променљиве не мења (на пример, ако је дужина низа током целог трајања програма иста), нећемо обраћати пажњу на разлику између променљиве програма (нпр. `n`) и њене вредности (нпр. n). Елементе низова ћемо такође обележавати индексима и обично ћемо претпостављати да бројање креће од нуле (нпр. a_0, a_1, ...).

Формално, можемо доказати следећу теорему.

Теорема: Ако је низ a дужине $n \geq 1$, непосредно пре почетка петље, у сваком кораку петље (и на њеном почетку, непосредно након провере услова, али и на њеном крају, непосредно након извршавања тела), као и након извршавања целе петље важи да је $1 \leq i \leq n$ и да је m минимум првих i елемената низа (где је i текућа вредност променљиве `i`, а m текућа вредност променљиве `m`).

Ово тврђење можемо доказати индукцијом и то по броју извршавања тела петље (обележимо тај број са k). Напоменимо само да ћемо петљу `for` сматрати само скраћеницом за петљу `while`, тако да ћемо иницијализацију петље сматрати за кôд који се извршава пре петље, док ћемо корак петље сматрати као последњу наредбу тела петље.

```
    int n = a.size();
    int m = a[0];
    int i = 1;
    while (i < n) {
        m = min(m, a[i]);
        i++;
    }
```

Такође, имплицитно ћемо подразумевати да се током извршавања петље низ ни у једном тренутку не мења (и то се експлицитно може доказати индукцијом). Ни променљива `n` не мења своју вредност.

Да бисмо у доказу били прецизнији, обележимо са $m_0, m_1, \ldots, m_k, \ldots$ вредности променљиве `m`, а са $i_0, i_1, \ldots, i_k, \ldots$ вредност променљиве `i` након $0, 1, \ldots, k, \ldots$ извршавања тела петље. Пошто променљива `n` не мења своју вредност, употребљаваћемо само ознаку n.

- Базу индукције чини случај $k = 0$ тј. случај када се тело петље није још извршило. Пре уласка у петљу променљива `i` се иницијализује на 1 (важи $i_0 = 1$). Пошто претпостављамо да је низ непразан, важи да је $1 \leq i = i_0 = 1 \leq n$. Променљива `m` се иницијализује на вредност `a[0]` (важи $m_0 = a_0$), што је заиста минимум једночланог префикса низа a. Дакле, услови су задовољени пре првог извршавања тела петље.

- Претпоставимо сада као индуктивну хипотезу да тврђење важи након k извршавања тела петље. Дакле,

претпостављамо да услови теореме важе за вредности m_k и i_k тј. да је $1 \leq i_k \leq n$ и да је m_k једнако минимуму првих i_k елемената низа (са i_k и m_k обележавамо вредности променљивих након k извршавања тела петље). Ако је услов петље испуњен, то ће уједно бити и вредности променљивих на почетку тела петље, пре њеног $k+1$-вог извршавања. Након k извршавања тела петље важи да је $i_k = k+1$, јер је променљива `i` имала почетну вредност 1 и тачно k пута је увећана за 1 (и ово би се формално могло доказати индукцијом).

Из индуктивне хипотезе и претпоставке да је услов петље `i < n` испуњен (тј. да је $i_k < n$) докажимо да након $k+1$ извршавања тела петље услови теореме важе и за вредности m_{k+1} и i_{k+1} (са m_{k+1} и i_{k+1} обележавамо вредности променљивих након $k+1$ извршавања тела петље). Вредности m_{k+1} и i_{k+1} се могу лако одредити на основу вредности m_k и i_k, анализом једног извршавања тела петље. Важи да је $i_{k+1} = i_k + 1 = k + 2$. Зато, пошто је $1 \leq i_k = k + 1 < n$, важи и да је $1 \leq i_{k+1} = k + 2 \leq n$, па је услов који се односи на распон вредности променљиве `i` очуван. Докажимо и да је m_{k+1} минимум првих i_{k+1} елемента низа. Важи да је m_{k+1} минимум вредности m_k и елемента a_{i_k}, тј. a_{k+1}. На основу индуктивне хипотезе знамо да је m_k минимум првих $i_k = k + 1$ елемената низа. Зато ће m_{k+1} бити минимум првих $k + 2$ елемената низа (закључно са елементом a_{k+1}), што је тачно i_{k+1} елемената низа, па и други услов остаје очуван.

Означимо са i и m вредности променљивих `i` и `m` након извршавања петље. На основу доказаног тврђења знамо да услови наведени у њему важе и након завршетка петље. Када се петља заврши, важи да је $i = n$ (јер на основу првог услова знамо да је $1 \leq i \leq n$, а услов петље `i < n` није испуњен). На основу другог услова знамо да је m минимум n чланова низа (што је заправо цео низ, јер је n његова дужина), тј. да променљива `m` садржи тражену вредност, чиме је доказана парцијална коректност. Заустављање се доказује једноставно тако што се докаже да се у сваком кораку петље ненегативна вредност $n - i$ смањује за по 1, док не постане 0.

Ако размотримо структуру претходног разматрања, можемо установити да смо идентификовали логичке услове који су испуњени непосредно пре и непосредно након сваког извршавања тела петље. Такви услови се називају **инваријанте петље**. Да бисмо доказали да је неки услов инваријанта петље, довољно је да докажемо:

(1) да тај услов важи пре првог уласка у петљу и

(2) да из претпоставке да тај услов важи пре неког извршавања тела петље и да је услов петље испуњен докажемо да тај услов важи и након извршавања тела петље.

Те две чињенице нам, на основу индуктивног аргумента, гарантују да ће услов бити испуњен пре и после сваке итерације петље, као и након извршавања целе петље (ако се она икада заустави), тј. да ће тај услов бити инваријанта петље (тај доказ се може спровести класичном математичком индукцијом на основу броја извршавања тела петље). Приметимо да први корак одговара доказивању базе индукције, а други доказивању индуктивног корака.

Свака петља има пуно инваријанти, међутим, од интереса су нам само оне инваријанте које у комбинацији са условом прекида петље (под претпоставком да петља није прекинута наредбом `break`) имплицирају услов који нам је потребан након петље. Ако је петља једина у неком алгоритму, обично је то онда услов коректности самог алгоритма. Дакле, након доказа леме која чини основу доказа да је неки услов инваријанта петље, потребно је да докажемо и

(3) да из тога да инваријанта важи након завршетка петље и да услов петље није испуњен следи коректност алгоритма.

Дакле, општа структура анализе програма коришћењем инваријанти се може описати на следећи начин.

```
<incijalizacija>
// ovde vazi <invarijanta>
while (<uslov>)
   // ovde vaze i <uslov> i <invarijanta>
   <telo>
   // ovde vazi <invarijanta>
// ovde ne vazi <uslov>, a vazi <invarijanta>
```

Изолујмо кључне делове претходног доказа и прикажимо их у формату који ћемо и убудуће користити приликом доказивања инваријанти петљи (индукција ће у тим доказима бити само имплицитна).

Лема: Ако је низ a дужине $n \geq 1$, услов да је $1 \leq i \leq n$ и да је m минимум првих i елемената низа је инваријанта петље (где са i обележавамо текућу вредност променљиве `i`, а са m текућу вредност променљиве

`m`).

- Пре уласка у петљу променљива `i` се иницијализује на 1 (важи $i = 1$). Пошто претпостављамо да је низ непразан, важи да је $1 \leq i \leq n$. Променљива `m` се иницијализује на вредност `a[0]` (важи $m = a_0$), што је заиста минимум једночланог префикса низа a.
- Претпоставимо да тврђење важи након уласка у петљу тј. да је вредност променљиве `m` (означимо је са m) једнака минимуму првих i чланова низа (где је i вредност променљиве `i` на уласку у петљу), да је $1 \leq i \leq n$, као и да је услов петље испуњен тј. да је $i < n$.

  Пошто је након извршавања тела петље вредност променљиве `i` увећана за један, важи да је $i' = i + 1$ (где са i' обележавамо вредност променљиве i након извршавања тела и корака петље). Пошто је важи да је $i < n$ и $1 \leq i \leq n$, након извршавања тела петље, важиће да је $1 \leq i' \leq n$.

  Нова вредност променљиве `m` (означимо је са m') биће једнака мањој од вредности m и a_i. На основу претпоставке важи да је m једнако минимуму првих i елемената низа, тј. минимуму бројева a_0, …, a_{i-1}, па је m' једнако минимуму бројева a_0, …, a_i, што је управо минимум првих $i+1$ елемената низа, па је заиста m' минимум првих i' елемената низа.

Теорема: Након извршавања петље, променљива `m` садржи минимум целог низа.

На основу инваријанте важи да је $1 \leq i \leq n$, а пошто по завршетку петље њен услов није испуњен, важи да је $i = n$. На основу инваријанте важи и да променљива `m` садржи минимум првих i елемената низа, а пошто је $i = n$, где је n број чланова низа, то је заправо минимум целог низа.

У наставку овог поглавља видећемо још неколико примера примене технике инваријанте петље. Мора се признати да када се техника користи потпуно формално, да би се доказала коректност већ написаног програмског кода, то не делује нарочито инспиришуће (поготово, ако су програми једноставни и ако је једноставно интуитивно разумети разлоге њихове коректности). Ретко када се у практичном програмирању коректност заиста доказује потпуно формално (осим у случају софтвера који може да угрози велики број живота, попут, на пример, софтвера који управља метро-системом у Паризу, који јесте у потпуности формално верификован). Међутим, аргументе и инваријанте на којима коректност почива програмер често "проврти по глави". Видећемо и да се техника инваријанти може употребити и пре него што је програм написан у циљу извођења програмског кода из спецификације. Јасне инваријанте често једнозначно указују на то како програмски код треба да изгледа и на тај начин помажу у процесу програмирања.

Задаци који су одабрани нису ни по чему посебни – они ће бити поновљени у поглављима у којима се уводе опште програмерске технике које се у њима примењују.

Задатак: Аритметички троугао

Колики је збир бројева у датом реду следећег троугла?

```
         1
      2  3  4
   5  6  7  8  9
10 11 12 13 14 15 16
        ...
```

Улаз: Редни број k ($1 \leq k \leq 5 \cdot 10^5$), реда троугла чији збир треба израчунати (бројање редова почиње од 1).

Излаз: Збир вредности у задатом реду троугла.

Пример

Улаз	*Излаз*
3	35

Решење

Итерација

До решења се може доћи коришћењем петљи. У првој петљи одређујемо први елемент k-тог реда, а уз то одређујемо и број елемената у њему. Први елемент одређујемо тако што саберемо број елемената у претходних $k - 1$ редова, док број елемената сваког реда тако што у сваком кораку број елемената претходног реда

увећавамо за два (сваки наредни ред има тачно два елемента више од претходног). Дакле, у петљи одржавамо почетак и број елемената текућег реда (иницијализујемо их на један, јер први ред почиње од један и има тачно један елемент) и $k-1$ пута почетак реда увећавамо за број елемената текућег реда, а број елемената текућег реда за два.

Прикажимо ово на примеру одређивања првог елемента реда 5.

```
k   pocetak   brojElemenata
1   1         1
2   2         3
3   5         5
4   10        7
5   17        9
```

Након тога, у другој петљи одређујемо збир елемената у k-том реду, тако што на збир који иницијализујемо на нулу додајемо један по један елемент тог реда, коришћењем алгоритма сабирања серије бројева – елементи су узастопни природни бројеви, па их је лако набројати.

У првој петљи обилазимо k редова којима одређујемо почетак помоћу две операције сабирања. Пажљивом анализом можемо закључити да у реду k има $2k-1$ елемената (мада ту чињеницу нисмо употребили у програму), који се у другој фази сабирају. Сложеност обе фазе, па и укупног решења је, дакле, $O(k)$. Задатак, наравно, може да се реши и у мањој сложености од ове, без коришћења петљи.

Докажимо формално коректност овог поступка. Инваријанта петље је да након m извршавања њеног тела важи да је $i = m+1$ и $1 \leq i \leq k$, као и да променљива `pocetak` садржи први елемент реда $i = m+1$, а променљива `brojElemenata` садржи број елемената реда $i = m+1$.

- Пошто се променљива `i` иницијализује на вредност 1, након $m = 0$ корака важи да је $1 = 0+1$ и $1 \leq 1 \leq k$. Пошто су и обе променљиве иницијализоване на вредност 1, након $m = 0$ корака петље, променљиве заиста садрже први елеменат и број елемената реда $i = m+1 = 0+1 = 1$.
- Претпоставимо да инваријанта важи након m корака петље и да је услов петље испуњен тј. да је $i < k$.

  Докажимо да тврђење важи и након $m' = m+1$ извршавања тела и корака петље. Пошто након извршавања корака важи $i' = i+1$, а пошто је на основу индуктивне хипотезе важило да је $i = m+1$, важи и да је $i' = m'+1$. Пошто је услов петље био испуњен, важило је $i < k$, па уз индуктивну претпоставку $1 \leq i \leq k$, важи $1 \leq i' \leq k$.

  Означимо са p и b вредности променљивих `pocetak` и `brojElemenata` на улазу у тело петље, а са p' и b' њихове нове вредности, након извршења тела и корака петље. Анализом додела у телу петље, јасно видимо да је $p' = p+b$ и $b' = b+2$. На основу претпоставке важи да је p први елемент реда $i = m+1$, а да је b број елемената реда $i = m+1$. Сабирањем првог елемента било ког реда у троугла и броја елемената тог реда троугла добија се први елемент наредног реда троугла. Дакле, важи да је $p+b$ први елемент реда $i+1 = m+2$ троугла, па, важи да је $p' = p+b$ први елемент реда $i' = i+1 = m+2 = m'+1$. На основу дефиниције троугла важи јасно и да сваки наредни ред има и два елемента више него претходни. Зато, пошто је b број елемената реда $m+1$, важи и да је $b' = b+2$ број елемената реда $i' = m'+1$.
- На крају петље услов није испуњен, па важи да је $i \geq k$. Уз услов $1 \leq i \leq k$, мора да важи да је $i = k$. На основу инваријанте знамо да променљива `pocetak` садржи број елемената реда $i = m+1 = k$. Дакле, петља је коректно извршила свој задатак и у променљиву `pocetak` сместила први елемент реда k.

На сличан начин се може формално доказати и коректност друге петље (инваријанта је да након m њених корака променљива `zbirRedaTrougla` садржи збир првих m елемената реда k, а да променљива i садржи вредност елемента на позицији m у том реду, ако се позиције броје од 0).

Приметимо да смо у овој петљи увели и нову променљиву којом смо регистровали број елемената у текућем реду троугла. То што смо ту променљиву имали на располагању нам је помоглно да ажурирамо вредност почетног елемента наредног реда, међутим, "кредит" који смо добили на почетку тела петље морали смо да вратимо на крају тако што смо морали да ажурирамо и вредност те променљиве (и тако је припремимо за наредну итерацију). Ова техника је позната под именом **ојачавање индуктивне хипотезе** и често се користи приликом конструкције алгоритама.

```
#include <iostream>
using namespace std;
```

```
int main() {
  long long k;
  cin >> k;

  // odredjujemo prvi broj u k-tom redu trougla
  long long pocetak = 1;
  long long brojElemenata = 1;
  for (int i = 1; i < k; i++) {
    pocetak += brojElemenata;
    brojElemenata += 2;
  }

  // odredjujemo zbir elemenata u k-tom redu trougla
  long long zbirRedaTrougla = 0;
  for (long long i = pocetak; i < pocetak + brojElemenata; i++)
    zbirRedaTrougla += i;

  cout << zbirRedaTrougla << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Тробојка

Написати програм који учитава низ целих бројева а затим га трансформише тако да елементи буду подељени у три дела у зависности од задатих вредности A и B. У првом делу су елементи мањи од задате вредности A (вредности из интервала $[-\infty, A)$), у другом елементи већи или једнаки задатој вредности A и мањи или једнаки задатој вредности B (вредности из интервала $[A, B]$), а у трећем елементи већи од задате вредности B (вредности из интервала $(B, +\infty)$). Није битно у ком се редоследу налазе елементи унутар делова. Учитати елементе у низ, а затим реорганизовати редослед елемената у том низу (не користити помоћне низове).

Улаз: У једној линији стандардног улаза налази се број елемената низа, N, а затим се, у наредној линији налазе елементи низа раздвојени размацима. У последње две линије се налазе цели бројеви A и B одвојени празнином, и при томе је $A < B$.

Излаз: Исписати елементе резултујућег низа на стандардни излаз (могуће је исписати елементе сваке од три групе у посебном реду, раздвојене размацима, а могуће је исписати и цео низ у једном реду или у више редова).

Пример

Улаз	*Излаз*
10	1 2
1 3 5 4 8 5 7 2 3 6	5 3 5 3
3	7 6 8
5	

Решење

Један пролаз кроз низ

Задатак можемо решити помоћу само једног пролаза кроз низ и то "у месту" тј. без коришћења помоћног низа. Алгоритам у наставку познат је под називом "Холандска застава тробојка" (енгл. Dutch national flag) и приписује се чувеном информатичару Дајкстри (енгл. Edsger W. Dijkstra).

Одржаваћемо три променљиве l, d и i и током петље наметнућемо да важи $0 \leq l \leq d \leq i \leq n$ и да важе следећи услови.

- У интервалу позиција $[0, l)$ налазиће се елементи мањи од A тј. бројеви из интервала $(-\infty, A)$,
- у интервалу позиција $[l, i)$ налазиће се елементи из интервала $[A, B]$,
- у интервалу позиција $[i, d)$ налазиће се елементи који још нису испитани,

- у интервалу позиција $[d, n)$ налазиће се елементи који су већи од B тј. елементи из интервала $(B, +\infty)$.

Дакле, одржавамо распоред `<<<===???>>>`, где су са `<` обележени елементи прве групе, са `=` елементи друге, а са `>` елементи треће групе.

Да би инваријанта важила пре уласка у петљу, јасно је да мора да важи да је $i = 0$ и $d = n$ (јер су сви елементи из интервала $[i, d) = [0, n)$ неиспитани). Такође, да бисмо били сигурни да су и интервалу $[0, l)$ сви елементи мањи од A, тај интервал мора бити празан и мора да важи да је $l = 0$. Након овакве иницијализације и интервал $[l, i) = [0, 0)$ и интервал $[d, n) = [n, n)$ је празан, па задовољава наметнути услов.

Петља ће се извршавати док год има неиспитаних елемената, а то је док је $i < d$. Размотримо како треба да изгледа тело петље, да би услови били одржани.

- Ако је елемент на позицији i мањи од броја A тада ћемо га заменити са елементом на позицији l (првим елементом из интервала $[A, B]$), након чега можемо увећати и i и l.
- У супротном, ако је елемент на позицији i мањи или једнак од B он припада интервалу $[A, B]$ и већ је на свом допуштеном месту, па само можемо увећати вредност i.
- У супротном елемент је већи од B и тада можемо смањити вредност d и разменити елемент на позицији i са елементом на (умањеној) позицији d, не мењајући вредност i (да би се елемент који је управо доведен на позицију i могао испитати у наредној итерацији).

На крају петље важи да је $i = d$. Уз остале наметнуте услове тврђење одатле следи (елементи из интервала позиција $[0, l)$ су мањи од A, елементи из интервала позиција $[l, i) = [l, d)$ су између A и B, интервал непрегледаних елемената $[i, d)$ је празан, док су елементи из интервала $[d, n)$ већи од B. Дакле, низ је разбијен на надовезане сегменте $[0, l)$, $[l, d)$ и $[d, n)$ и у сваком сегменту се налазе одговарајући елементи.

Размотримо рад алгоритма на једном примеру. Нека је $A = 4$, $B = 7$ и нека низ има садржај `5 1 8 6 3 9 4 2`. У наставку ћемо приказати стање низа током извођења алгоритма.

```
l             d
5 1 8 6 3 9 4 2
i

l             d
5 1 8 6 3 9 4 2
  i

  l           d
1 5 8 6 3 9 4 2
    i

  l         d
1 5 2 6 3 9 4 8
    i

    l       d
1 2 5 6 3 9 4 8
      i

    l       d
1 2 5 6 3 9 4 8
        i

      l     d
1 2 3 6 5 9 4 8
          i

      l   d
1 2 3 6 5 4 9 8
          i
```

```
     l     d
1 2 3 6 5 4 9 8
           i
```

У сваком кораку петље се или увећава `i` или смањује `d`, док се не сусретну, што се дешава у n корака. Сложеност овог приступа је, дакле, $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// funkcija ucitava elemente u vektor
vector<int> unosNiza() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  return a;
}

// funkcija organizuje elemente vektora tako da se prvo nalaze elementi
// za koje vazi da su iz intervala (-Inf, A), nakon toga dolaze
// elementi iz intervala [A, B], i nakon toga elementi iz intervala
// (B, Inf)
void podelaNiza(vector<int>& niz, int A, int B) {
  // - u intervalu pozicija [0, l) su elementi iz intervala (-Inf, A)
  // - u intervalu pozicija [l, i) su elementi iz intervala [A, B]
  // - u intervalu pozicija [i, d) su jos neispitani elementi
  // - u intervalu pozicija [d, n) su elementi iz intervala (B, Inf)
  int l = 0, i = 0, d = niz.size();
  // dok god postoje neispitani elementi
  while (i < d) {
    if (niz[i] < A)
      // menjamo tekuci element sa prvim elementom iz intervala [A, B]
      swap (niz[i++], niz[l++]);
    else if (niz[i] <= B)
      // tekuci element ostaje na svom mestu
      i++;
    else
      // menjamo tekuci element sa poslednjim neispitanim
      swap(niz[i], niz[--d]);
  }
}

// ispis elemenata vektora na standardni izlaz
void ispisNiza(const vector<int>& a, int A, int B) {
  int i = 0;
  // ispisujemo elemente iz intervala (-Inf, A)
  while (i < a.size() && a[i] < A)
    cout << a[i++] << " ";
  cout << endl;
  // ispisujemo elemente iz intervala [A, B]
  while (i < a.size() && a[i] <= B)
    cout << a[i++] << " ";
  cout << endl;
  // ispisujemo elemente iz intervala (B, +Inf)
```

```
  while (i < a.size())
    cout << a[i++] << " ";
  cout << endl;
}


int main()  {
  // ucitavamo elemente niza
  vector<int> a = unosNiza();
  // ucitavamo interval [A, B]
  int A, B;
  cin >> A >> B;
  // reorganizujemo elemente po intervalima (-inf, A), [A, B] i [B, inf)
  podelaNiza(a, A, B);
  // ispisujemo elemente niza
  ispisNiza(a, A, B);
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Двобојка

Напиши програм који организује елементе низа тако да прво иду сви парни елементи, а затим непарни, при чему међусобни редослед парних и непарних елемената није битан. Елементе прво учитати у низ, а затим тај низ трансформисати у линеарном времену (само једним пролазом кроз низ).

Улаз: У првој линији стандардног улаза унети природан број n ($1 \leq n \leq 50000$) - број елемената низа, а у наредној линија унети n природних бројевава у границама од 1 до 1000.

Излаз: На станардни излаз исписати елементе низа уређене на тражени начин, раздвојене са по једним размаком.

Пример

Улаз	*Излаз*
10 2 5 3 6 1 8 9 10 11 4	2 6 8 10 4 5 3 1 9 11

Задатак: Први који није дељив

Размотримо низ бројева $210, 2310, 390, 30, 510, 66, 6, 138, 46, 106, 59, 17, 23$. Он је интересантан из неколико разлога. На пример, првих пет бројева је дељиво са 10, а после ниједан број није дељив са 10. Првих десет бројева је парно, а после су сви бројеви непарни. Првих осам бројева је дељиво са 6, а после ниједан број није дељив са 6. Прва два броја су дељива са 210, а после ниједан број није дељив са 210, итд. Покушај да пронађеш још оваквих правилности. Напиши програм који за сваки унети делилац одређује колико бројева је дељиво њиме. Сматрати да за сваки унети делилац важи наведена правилност.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим у наредном реду n природних бројева (мањих од 10^{18}) раздвојених по једним размаком. Након тога се до краја улаза уносе делиоци (сваки у посебном реду). За сваки делилац се сигурно зна (и то није потребно проверавати) да се у низу налазе прво бројеви који јесу, а затим бројеви који нису дељиви тим делиоцем.

Излаз: За сваки унети делилац у посебном реду исписати број елемената низа који су њиме дељиви.

Пример

Улаз	*Излаз*
13	5
210 2310 390 30 510 66 6 138 46 106 59 17 23	10
10	8
2	10
6	0
2	5
4	
15	

Решење

Бинарна претрага

Захваљујући интересантној особини низа, задатак ефикасно може бити решен применом алгоритма бинарне претраге. У питању је варијанта алгоритма бинарне претраге у ком се уместо позиције конкретне вредности у сортираном низу захтева проналажење прве позиције на којој се налази елемент који задовољава неки услов. Наиме, под претпоставком да се у низу прво налазе елементи који не задовољавају тај услов, а затим елементи који задовољавају тај услов, *преломну тачку* (тренутак када се из једне прелази у другу групу елемената) можемо наћи бинарном претрагом. Дакле, ако је низ облика `-------+++++++`, бинарном претрагом можемо пронаћи позицију последњег минуса, првог плуса, број минуса или број плусева, где смо са `-` означили оне елементе који не задовољавају, а са `+` оне елементе који задовољавају дати услов.

Ручно имплементирана бинарна претрага

Током рада алгоритма, одржавамо две променљиве l и d такве да важи инваријанта да је $0 \leq l \leq d+1 \leq n$ и да су

- лево од l тј. у интервалу позиција $[0, l)$ елементи који не задовољавају услов,
- десно од d тј. у интервалу позиција (d, n) елементи који задовољавају услов.

У интервалу позиција $[l, d]$ налазе се елементи чији статус још није познат. На почетку су сви елементи непознати, па је јасно да интервал $[l, d]$ треба иницијализовати на $[0, n-1]$, тј. променљиву l треба иницијализовати на нулу, а d на вредност $n-1$. Интервали $[0, l)$ и (d, n) су празни, па је инваријанта очувана (услов $l \leq d+1$ се своди на $0 \leq n$, што је тривијално испуњено).

Ако интервал позиција $[l, d]$ није празан тј. ако је $l \leq d$, проналазимо му средину $s = l + \lfloor \frac{d-l}{2} \rfloor$.

- Ако елемент на позицији s задовољава услов, тада на основу монотоности услов задовољавају и сви елементи десно од s. Зато померамо d за једно место лево од средине тј. вредност променљиве d постављамо на $s-1$.
- Ако елемент на позицији s не задовољава услов, тада на основу монотоности услов не задовољавају ни сви елементи лево од s. Зато померамо l за једно место десно од средине тј. вредност променљиве l постављамо на $s+1$.

Претрага траје све док се интервал $[l, d]$ не испразни, тј. док је $l \leq d$. Тада је $l = d+1$ и елементи који не задовољавају услов се налазе на позицијама $[0, l) = [0, d]$, док се елементи који не задовољавају услов налазе на позицијама $(d, n) = [d+1, n) = [l, n)$. Дакле, први елемент који задовољава услов је на позицији l, а последњи који не задовољва услов на позицији d.

Прикажимо рад алгоритма на једном примеру.

```
l               d
1 7 3 5 9 11 2 8 6
      s

        l     d
1 7 3 5 9 11 2 8 6
          s

        ld
1 7 3 5 9 11 2 8 6
```

```
          s

          d  l
1 7 3 5 9 11 2 8 6
```

Докажимо формално коректност овог алгоритма. Уз поменуте услове инваријанта је и да важи $0 \leq l \leq d+1 \leq n$.

Након иницијализације $l = 0$, $d = n-1$, услови су испуњени (интервали $[0, l)$ и (d, n) су празни, док је услов $0 \leq l \leq d+1 \leq n$ еквивалентан услову $0 \leq 0 \leq n \leq n$ и тривијално је испуњен.

Ако интервал позиција $[l, d]$ није празан тј. ако је $l \leq d$, проналазимо му средину $s = l + \lfloor \frac{d-l}{2} \rfloor$. Пошто је $l \leq d$, важи и да је $l \leq s \leq d$.

- Ако елемент на позицији s задовољава услов, тада на основу монотоности услов задовољавају и сви елементи десно од s. Зато вредност промењиве d постављамо на $s-1$ (нове вредности промењивих су $l' = l$ и $d' = s-1$). Тиме инваријанта остаје на снази (посебно, сви елементи у интервалу позиција $(s-1, n) = [s, n)$ задовољавају услов). Важи и услов $0 \leq l' \leq d'+1 \leq n$, јер је он еквивалентан услову $0 \leq l \leq s \leq n$.
- Ако елемент на позицији s не задовољава услов, тада на основу монотоности услов не задовољавају ни сви елементи лево од s. Зато вредност промењиве l постављамо на $s+1$ (нове вредности промењивих су $l' = s+1$ и $d' = d$). Тиме инваријанта остаје одржана (посебно, ниједан елемент у интервалу позиција $(0, l) = [0, s]$ не задовољава услов). Важи и услов $0 \leq l' \leq d'+1 \leq n$ који је еквивалентан услову $0 \leq s+1 \leq d+1 \leq n$.

Када се интервал испразни, тада је $l > d$, па пошто важи $0 \leq l \leq d+1 \leq n$, важи и $l = d+1$. На основу инваријанте знамо да су елементи који задовољавају услов на позицијама $(d, n) = [l, n]$. Зато је први елемент који задовољава услов је на позицији l (што је уједно и број елемената који не задовољавају услов). Елементи који не задовољавају услов су на позицијама $[0, l) = [0, d+1) = [0, d]$, па је последњи елемент који не задовољава на позицији d.

Заустављање се лако доказује тако што се доказује да се у сваком кораку петље интервал $[l, d]$ тј. његова дужина $d-l+1$ смањује, што је прилично очигледно и када је $l' = l$ и $d' = s-1 < d$ и када је $l < l' = s+1$ и $d' = d$.

Пошто се у сваком кораку претраге ширина интервала $[l, d]$ преполови, пошто се иницијално креће од интервала $[0, n-1]$ који има n елемената и пошто се алгоритам завршава када се интервал испразни, сложеност алгоритма је $O(\log n)$. Наиме, дужина интервала после k корака је $\lfloor \frac{n}{2^k} \rfloor$ и важи да је $\lfloor \frac{n}{2^k} \rfloor < 1$ када је $k > \log_2 n$.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long k;
  while (cin >> k) {
    int l = 0, d = n-1;
    while (l <= d) {
      int s = l + (d - l) / 2;
      if (a[s] % k != 0)
        d = s - 1;
      else
```

```
        l = s + 1;
    }
    cout << l << '\n';
  }
  return 0;
}
```

Исправљање грешака на основу формалне анализе кода

Када је кôд коректан, доказ је обично неинформативан. Помаже нам да "мирно спавамо", али ништа више од тога. Много интересантнија ситуација се дешава у случају када нам формално резоновање о коду помаже да детектујемо и исправимо грешке у програму (тзв. багове). Погледајмо наредни покушај имплементације алгоритма.

```
int l = 0, d = n;
while (l < d) {
  int s = l + (d - l) / 2;
  if (a[s] % k != 0)
    d = s-1;
  else
    l = s+1;
}
cout << d+1 << '\n';
```

На основу инцијализације делује да покушавамо да претражимо полузатворени интервал $[l, d)$. Пошто је у питању бинарна претрага, изгледа да се намеће инваријанта да је $0 \leq l \leq d \leq n$ и да су:

- сви елементи из $[0, l)$ дељиви са k,
- ниједан елемент из интервала $[d, n)$ није дељив са k.

На почетку су оба та интервала празна, па инваријанта за сада добро функционише. Ако погледамо услов петље, делује да петља ради док се интервал непознатих елемената $[l, d)$ не испразни (заиста, када је $l \geq d$, тај интервал је празан). За сада све ради како треба. Покушамо сада да проверимо да ли извршавање тела петље одржава инваријанту.

- Ако је a_s није дељив са k, тада се променљива `d` поставља на вредност $d' = s - 1$. На основу инваријанте треба да важи да ниједан елемент у интервалу $[d', n)$ није дељив са k. Међутим, ми то не знамо, јер само знамо да је a_s није дељив са k, али не знамо да a_{s-1} није дељив са k. Дакле, овде се сигурно крије грешка у коду. Ако доделу `d = s-1` заменимо са `d = s`, тада ће инваријанта бити одржана (јер знамо да a_s није дељив са k, па са k неће бити дељив ниједан елемент иза њега).
- Ако a_s јесте дељив са k, тада се променљива `l` поставља на вредност $l' = s + 1$. На основу инваријанте треба да важи да су сви елементи у интервалу $[0, l')$ дељиви са k, међутим, то ће овде бити испуњено, јер је a_s дељив са k, па су са k дељиви и сви елементи испред њега. Дакле, у овом случају је кôд коректан и инваријанта остаје одржана.

На крају, када се петља заврши можемо закључити да важи да је $l = d$ (јер све време важи да је $l \leq d$, а након петље не важи да је $l < d$). У коду се за позицију првог елемента који није дељив са k проглашава позиција $d + 1$. Иако је у оригиналној варијанти кода `l` могло без проблема да се замени са `d+1`, у овој варијанти то није могуће. Наиме, ми на основу инваријанте овог кода знамо да се на позицији $l = d$ налази елемент који није дељив са k, а да се на позицији $l - 1$ налази елемент који јесте дељив са k (осим када је $l = 0$ и тада нема елемената дељивих са k). Зато крајњи резултат није коректан и потребно га је заменити са `d`, јер се први елемент који није дељив са k налази на позицији d (осим када су сви елементи дељиви са k, када је $d = n$, но и тада је d исправна повратна вредност). Дакле, формалном анализом смо открили и исправили две грешке.

Програмери често програм исправљају тако што насумице покушавају да помере индексе за 1 лево или десно, да замене мање са мање или једнако и слично. Већ на овако кратким програмима се види да је простор могућих комбинација велики, а да је могућност за грешку приликом таквог експерименталног приступа веома велика. Стога је увек боље застати, формално анализирати шта је потребно да кôд ради и исправити га на основу резултата формалне анализе.

На крају, скренимо пажњу на још један детаљ исправљеног програма. Парцијална коректност је јасна на основу анализе коју смо спровели, међутим, заустављање може бити доведено у питање, с обзиром на наредбу

`d = s`. Заустављање доказујемо тако што показујемо да се у сваком корају смањује број непознатих елемената, тј. да дужина интервала $[l, d)$ која је једнака $d - l$ у сваком кораку петље опада. Пошто је $l \leq d$ инваријанта, смањивање не може трајати довека, па се у неком тренутку програм зауставља. Поставља се питање да ли се $d - l$ смањује и у измењеном коду у коме се јавља наредба `d=s`. Одговор је потврдан, а образложење је суптилно. Прво, на основу услова петље важи да је $l < d$. Даље, вредност s се израчунава наредбом `s = l + (d - l) / 2` што нам да је $s = \lfloor \frac{l+d}{2} \rfloor$. Због заокруживања наниже, важи да је $s < d$ и зато се након одређивања $d' = s$, $l' = l$ вредност $d' - l'$ смањује у односу на $d - l$. Важи и да је $l \leq s$, али пошто је у другој грани $l' = s + 1$ и $d' = d$, вредност $d' - l'$ се опет смањује у односу на $d - l$. Да је заокруживање којим случајем вршено навише (нпр. `s = l + (d - l + 1) / 2`), програм би могао упасти у бесконачну петљу.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long k;
  while (cin >> k) {
    int l = 0, d = n;
    while (l < d) {
      int s = l + (d - l) / 2;
      if (a[s] % k != 0)
        d = s;
      else
        l = s + 1;
    }
    cout << d << '\n';
  }
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Најмањи број који није збир елемената скупа

Дат је скуп природних бројева (задат у облику сортираног низа). Одредити најмањи природан број који није збир неких елемената тог скупа (сваки елемент скупа може само једном учествовати у збиру).

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^3$), а затим у наредном реду сортиран низ од n различитих природних бројева мањих од 10^4.

Излаз: На стандардни излаз исписати тражени најмањи природан број који није збир неких елемената тог скупа.

Пример

Улаз	*Излаз*
`8` `1 2 4 7 15 32 35 48`	`30`

Решење

Чињеница да су елементи сортирани олакшава решење задатка. Обрађиваћемо елемент по елемент и одржаваћемо границу до које смо сигурни да се сваки број може представити као збир неког подскупа. Можда

мало изненађујуће, та граница је у сваком кораку једнака збиру свих тренутно учитаних елемената. Ако је нови учитани елемент строго већи од збира свих претходних елемената увећаног за један, онда се тај увећани збир не може добити као подскуп. У супротном можемо бити сигурни да се сви бројеви од 0 па до збира свих елемената (у који је укључен и нови елемент) могу добити као збир неког подскупа. Наиме, пошто је у претходном кораку било могуће добити све бројеве од 1 до збира свих елемената без тог новог, када у све те подскупове укључимо нови елемент добићемо све бројеве од тог новог елемента, па до збира свих елемената са тим новим елементом.

На пример, нека је дат низ $1, 2, 3, 5, 14, 20, 27$.

- 0 можемо добити као збир празног скупа $\{\}$.
- 1 можемо добити као збир скупа $\{1\}$.
- Када у претходне скупове укључимо и 2, можемо добити све бројеве закључно са 3 (2 као $\{2\}$ и 3 као $\{1, 2\}$).
- Када у претходне скупове укључимо и 3, можемо добити све бројеве закључно са 6 (4 као $\{1, 3\}$, 5 као $\{2, 3\}$ и 6 као $\{1, 2, 3\}$).
- Када у претходне скупове укључимо 5 можемо добити све бројеве закључно са 11 (7 као $\{2, 5\}$, 8 као $\{1, 2, 5\}$ и 9 као $\{1, 3, 5\}$, 10 као $\{2, 3, 5\}$ и 11 као $\{1, 2, 3, 5\}$).
- Пошто је наредни број 14, јасно је да се број 12 не може никако добити.

Програм се решава једним проласком кроз низ бројева и сложеност је прилично очигледно $O(n)$.

Докажимо и формално коректност, овог алгоритма тј. програма датог у прилогу.

Лема: Нека је m вредност промењљиве `mozeDo`. Инваријанта петље је да је $0 \leq i \leq n$, да је m збир првих i елемената низа и да се сваки број из интервала $[0, m]$ може добити као збир неког подскупа првих i елемената низа.

Пре уласка у петљу је $i = 0$ и $m = 0$. Збир првих $i = 0$ елемената низа је по дефиницији нула (тј. m). Број 0 је једини елемент интервала $[0, m] = [0, 0]$ и он се може добити као збир празног подскупа (тј. 0 елемената полазног низа).

Претпоставимо да тврђење важи пре уласка у петљу.

- Ако је $a_i > m + 1$, тврдимо да је $m + 1$ тражени најмањи број. На основу инваријанте знамо да су сви бројеви из интервала $[0, m]$ покривени, тако да мањи број од $m + 1$ не може бити решење. Докажимо да број $m + 1$ не може бити збир подскупа. Пошто је низ сортиран, сви елементи од a_i до a_{n-1} су строго већи од $m+1$. Дакле, ни један од тих елемената не сме бити укључен у подскуп јер би њиховим укључивањем збир већ премашио $m + 1$. Подскуп се мора састојати само од елемената a_0 до a_{i-1}, међутим, пошто је m њихов збир, збир сваког њиховог подскупа је мањи или једнак m. Дакле, $m + 1$ се не може постићи и он је тражено решење.
- Ако је $a_i \leq m + 1$, тада је $m' = m + a_i$, $i' = i + 1$ и тврдимо да је m' збир свих елемената a_0, …, a_i и да се сваки број из интервала $[0, m']$ може представити као збир неког подскупа првих $i' = i + 1$ елемената низа. Прва тврдња је прилично очигледна, јер је по претпоставци m збир свих елемената a_0, …, a_{i-1}, а $m' = m + a_i$. На основу претпоставке знамо да сви бројеви из $[0, m]$ могу бити збирови подскупова првих i елемената низа. Слично и сви бројеви из интервала $[a_i, a_i + m]$ се могу добити као збир неког подскупа првих $i' = i + 1$ елемента низа. Наиме, тај подскуп ће бити унија елемента a_i и оног подскупа првих i елемената низа чији је збир једнак разлици између тог броја и броја a_i – он је из $[0, m]$, па на основу претпоставке такав подскуп постоји. Пошто је $a_i \leq m + 1$ унија интервала $[0, m]$ и $[a_i, a_i + m]$ је $[0, a_i + m] = [0, m']$. Зато је сваки елемент из $[0, m']$ једнак збиру неког подскупа првих i' елемената низа, па инваријанта остаје очувана.

Теорема: Случај када се петља заврши прекидом, јер је $a_i > m+1$ је већ размотрен. Када се петља заврши, важи да је $i = n$. На основу инваријанте m је збир свих елемената низа, и сваки број из $[0, m]$ јесте збир неког подскупа првих $i = n$ елемената низа, тј. целог низа. Зато је $m + 1$ најмањи елемент који није могуће добити (јер се укључивањем свих елемената добија највише m) и исписано решење је исправно.

```
#include <iostream>

using namespace std;

int main() {
  int n;
```

```
  cin >> n;
  // sabiranjem elemenata trenutnog sukpa mogu se dobiti svi elementi
  // iz intervala [0, mozeDo]
  int mozeDo = 0;
  for (int i = 0; i < n; i++) {
    int x; cin >> x;
    if (x > mozeDo + 1)
      break;
    mozeDo += x;
  }
  cout << mozeDo + 1 << endl;
  return 0;
}
```

Задатак: Бинарни запис

Напиши програм који на основу неозначеног целог броја n формира и исписује његов 32-битни бинарни запис.

Улаз: Са стандардног улаза се уноси број n ($0 \leq n \leq 2^{32} - 1$).

Излаз: На стандардни излаз исписати 32-битни бинарни запис броја n.

Пример 1

Улаз	*Излаз*
123456	00000000000000011110001001000000

Пример 2

Улаз	*Излаз*
16777215	00000000111111111111111111111111

Задатак: Растављање на просте чиниоце

Ако је дато неколико простих бројева, њихов производ се може веома лако и брзо одредити. Међутим, ако је дат производ, често је веома тешко одредити просте бројеве који га сачињавају. Напиши програм који што ефикасније решава тај проблем.

Улаз: Са стандардног улаза се уноси један природан број n ($1 \leq n \leq 2 \cdot 10^9$).

Излаз: На стандардни излаз исписати просте чиниоце броја n, уређене од најмањих до највећих, раздвојене размаком.

Пример

Улаз	*Излаз*
900	2 2 3 3 5 5

Решење

Алгоритам факторизације

Потенцијални чиниоци f броја n се испитују редом, у петљи, кренувши од броја 2. У сваком кораку испитује се да ли је број n дељив бројем f и док год јесте дељив, у унутрашњој петљи, он се дели бројем f приј-ављујући при том чинилац f (пошто се у склопу услова унутрашње петље врши провера дељивости n са f, није потребна посебна провера дељивости наредбом гранања пре те петље). Након тога прелази се на следећи потенцијални чинилац (за један већи од претходног). Иако се може помислити да је за сваки потенцијални чинилац потребно проверити да ли је он прост број (јер нас занимају само прости чиниоци), то није потребно радити. Наиме, у поступку претраге који смо навели, ако текући кандидат f није прост, он не може да дели број n, јер смо све његове просте чиниоце већ дељењем уклонили из броја n. На пример, када f достигне вредност 6, број n не може бити дељив њиме јер је претходно исцрпно издељен бројем 2 (а касније и бројем 3). Заиста, ако претпоставимо супротно да f дели n и да је f сложен број, тада би f имао неки прости чинилац мањи од њега и то би уједно био прост чинилац броја n. Међутим, то није могуће јер смо пре увећања броја f на његову текућу вредност утврдили да текући број n не може бити дељив ни једним бројем мањем од f

(иначе бисмо га делили са f, а не увећавали f). Дакле сложени чиниоци се елиминишу тако што се утврди да текући број n није дељив њима, што је много ефикасније него примењивати на њих тест простости.

У најједноставнијој имплементацији, описани поступак траје све док се број n дељењем својим чиниоцима не сведе на број 1.

Иако коректан, алгоритам који се завршава свођењем броја на 1 је прилично неефикасан и за бројеве који имају велике просте чиниоце ради веома споро (покушајте извршавање програма нпр. за број 1000000007). Проблем настаје јер се делиоци последњег простог чиниоца испитују све док се не дође до самог тог броја. С обзиром на ограничење бројевног типа, број чинилаца можемо сматрати практично константним (не може их бити више од 32), па је сложеност $O(M)$, где је M највећи прост фактор полазног броја.

На срећу, алгоритам је једноставно поправити, тако што се растављање заустави чим се утврди да је текућа вредност промењиве n прост број (а видећемо да за то није потребно чекати да вредност f достигне n).

Прикажимо рад алгоритма на једном примеру.

```
n       f       cinioc
3300    2       2
1650    2       2
825     2       -
825     3       3
275     3       -
275     4       -
275     5       5
55      5       5
11      5       -
11      6       -
11      7       -
11      8       -
11      9       -
11      10      -
11      11      11
1       11      -
```

Докажимо коректност претходног алгоритма и формално, коришћењем технике инваријанти петље. Централна инваријанта петље је то да текућа вредност промењиве n није дељива ни једним бројем из интервала $[2, d)$, као и да је почетни број n_0 производ до сада исписаних простих бројева и текуће вредности промењиве n.

Пре уласка у петљу је $d = 2$, па је интервал $[2, d)$ празан и први део инваријанте тривијално важи. Важи и да је $n = n_0$ и да ниједан број није исписан, па и други део инваријанте важи.

Претпоставимо да инваријанта важи на уласку у петљу.

- Ако је n дељив бројем d, исписује се број d. Он је очигледно чинилац броја n. Претпоставимо да је сложен и да се може раставити као $d = d_1 \cdot d_2$, за $d_1 > 1$ и $d_2 > 1$. Тада би број n био дељив са d_1 и са d_2 који припадају интервалу $[2, d)$, што је супротно инваријанти, па закључујемо да d мора бити прост. Дељењем броја n са d добија се нови број n', који такође није дељив ни са једним бројем из $[2, d)$, па први део инваријанте остаје очуван. Почетна вредност n_0 остаје једнака производу исписаних бројева и текуће вредности n (јер је то важило на основу инваријанте, исписано је d, а n је подељено са d), па и други део инваријанте остаје очуван.
- Ако n није дељив бројем d, тада се d увећава за 1 (нека је $d' = d + 1$). Да би први део инваријанте био одржан, треба да важи да n није дељив ни са једним бројем из интервала $[2, d') = [2, d]$, но то важи, јер на основу инваријанте знамо да n није дељив ни са једним бројем из интервала $[2, d)$, а на основу експлицитне провере услова знамо да n није дељиво ни са d. Ниједан број није исписан, нити је број n промењен, па други део инваријанте тривијално наставља да важи.

На основу инваријанте знамо да је n_0 једнак производу свих исписаних простих чинилаца и тренутне вредности броја n. Пошто је када се алгоритам заврши она једнака 1, исписана је проста факторизација броја n.

```
#include <iostream>
```

```cpp
using namespace std;

int main() {
  // ucitavamo broj koji treba rastaviti na proste cinioce
  int n;
  cin >> n;
  int f = 2; // prvi potencijalni prost cinilac je 2
  // dok se broj deljenjem sa svojim prostim ciniocima ne svede na 1
  while (n > 1) {
    while (n % f == 0) {   // dok je n deljivo sa f
      cout << f << " ";    //   prijavljujemo pronađeni prost cinilac
      n /= f;              //   delimo broj njime
    }
    f = f + 1;             //   prelazimo na sledeceg kandidata
  }
  cout << endl;
  return 0;
}
```

Глава 2

Сложеност израчунавања

Важно питање за практичну примену написаних програма је то колико ресурса програм захтева за своје извршавање. Најважнији ресурси су сигурно време потребно за извршавање програма и заузета меморија, мада се могу анализирати и други ресурси (на пример, код мобилних уређаја важан ресурс је утрошена енергија). Дакле, обично се разматрају:

- **временска** сложеност алгоритма;
- **просторна (меморијска)** сложеност алгоритма.

На пример, ако један програм израчунава потребан број за 10 секунди, а други за два и по минута, јасно је да је први програм практично примењивији. Међутим, ако први програм за своје извршавање захтева преко 10 гигабајта меморије, други око 1 гигабајт, а ми имамо рачунар са 4 гигабајта меморије, први програм нам је практично неупотребљив (иако ради много брже од другог). Ипак, с обзиром на то да савремени рачунарски системи имају прилично велику количину меморије, време је чешће ограничавајући фактор и у наставку ћемо се чешће бавити анализом временске ефикасности алгоритама.

При том, прилично је релативно колико брзо програм треба да ради да бисмо га сматрали ефикасним. На пример, ако програм успе да за пола сата реши неки нерешен математички проблем, који људи годинама нису могли да реше, он је свакако користан и можемо га сматрати веома ефикасним. Са друге стране, ако програм уграђен у аутомобил контролише кочнице приликом проклизавања, њему и неколико стотина милисекунди израчунавања може бити превише, јер ће за то време аутомобил неконтролисано слетети са пута.

Понашање програма (па и количина утрошених ресурса), наравно, зависи од његових улазних параметара. Јасно је, на пример, да ће програм брже израчунати просечну оцену двадесетак ученика једног одељења, него просечну оцену неколико десетина хиљада ученика који полажу Државну матуру. Такође се може претпоставити да понашање програма не зависи од конкретних оцена које су ученици добили, већ само од броја ученика. Зато сложеност алгоритма често изражавамо у функцији *величине (димензије) његових улазних параметара*, а не самих вредности параметара. Величина улазне вредности може бити број улазних елемената које треба обрадити, број битова потребних за записивање улаза који треба обрадити, сам улазни број који треба обрадити итд. Увек је потребно експлицитно навести у односу на коју величину улазне вредности се разматра сложеност.

Са друге стране, неки се алгоритми не извршавају исто за све улазе исте величине, па је потребно наћи начин за описивање ефикасности алгоритма на разним могућим улазима исте величине.

- **Анализа најгорег случаја** заснива процену сложености алгоритма на најгорем случају (на случају за који се алгоритам најдуже извршава – у анализи временске сложености, или на случају за који алгоритам користи највише меморије – у анализи просторне сложености). Та процена може да буде варљива, тј. превише песимистична. На пример, ако се програм у 99,9% случајева извршава испод секунде, док се само у 0,1% случајева извршава за око 10 секунди, анализом најгорег случаја закључили бисмо да ће се програм извршавати за око 10 секунди. Са друге стране, анализа најгорег случаја нам даје јаке гаранције да програм који је у најгорем случају довољно ефикасан у свим случајевима може да се изврши са расположивим ресурсима.
- У неким ситуацијама могуће је извршити **анализу просечног случаја** и израчунати просечно време извршавања алгоритма, али да би се то урадило, потребно је прецизно познавати простор допуштених

улазних вредности и вероватноћу да се свака допуштена улазна вредност појави на улазу програма. У случајевима када је битна гаранција ефикасности сваког појединачног извршавања програма процена просечног случаја може бити варљива, превише оптимистична, и може да се деси да у неким ситуацијама програм не може да се изврши са расположивим ресурсима. На пример, анализа просечног случаја би за претходни програм пријавила да се у просеку извршава испод једне секунде, међутим, за неке улазе он се може извршавати и преко десет секунди.

- Анализа најбољег случаја је, наравно, превише оптимистична и никада нема смисла.

Некада се анализа врши тако да се процени укупно време потребно да се изврши одређен број сродних операција. Тај облик анализе назива се **амортизована анализа** и у тим ситуацијама нам није битно време извршавања појединачних операција, већ само збирно време извршавања свих операција.

У наставку ће, ако није другачије речено другачије, бити подразумевана анализа најгорег случаја.

2.1 Мерење времена извршавања

Понашање за конкретне вредности улазних параметара се може експериментално одредити, тестирањем рада програма. На пример, размотримо наредна два алгоритма (дата у псеудокоду) који израчунавају збирове природних бројева од 1 до i, за свако i из интервала 1 до n.

```
zbir = 0
foreach i in [1, n]:
    zbir += i
    print(zbir)
```

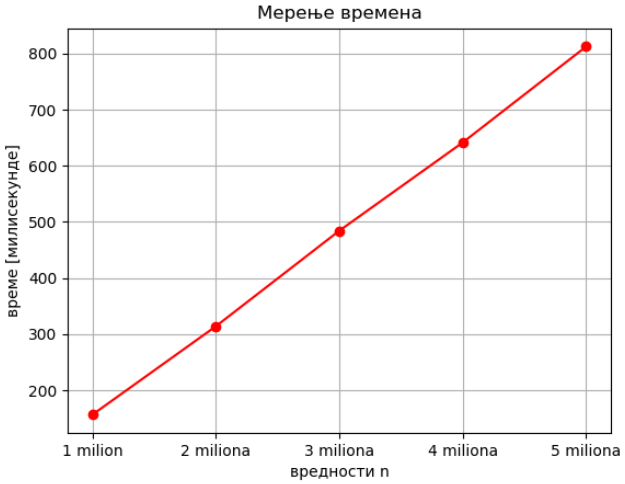
и

```
function zbir(k):
    zbir = 0
    foreach i in [1, k]:
        zbir += i
    return zbir

foreach k in [1, n]:
    print(zbir(k))
```

Ако се први алгоритам имплементира (на пример, у програмском језику Python) и ако се измери његово време извршавања за различите вредности n, добијају се следећи резултати (као резултат приказан је збир свих бројева од 1 до n).

```
n =  1000000, rezultat:   500000500000, vreme: 0.16 sekundi
n =  2000000, rezultat:  2000001000000, vreme: 0.31 sekundi
n =  3000000, rezultat:  4500001500000, vreme: 0.49 sekundi
n =  4000000, rezultat:  8000002000000, vreme: 0.65 sekundi
n =  5000000, rezultat: 12500002500000, vreme: 0.83 sekundi
```

Већ одавде се види да су времена приближно сразмерна вредностима n. Још прегледнији начин да уочимо ову пропорционалност је приказивање графика времена израчунавања суме у зависности од n, са кога се јасно види да код првог програма време извршавања приближно линеарно зависи од n.

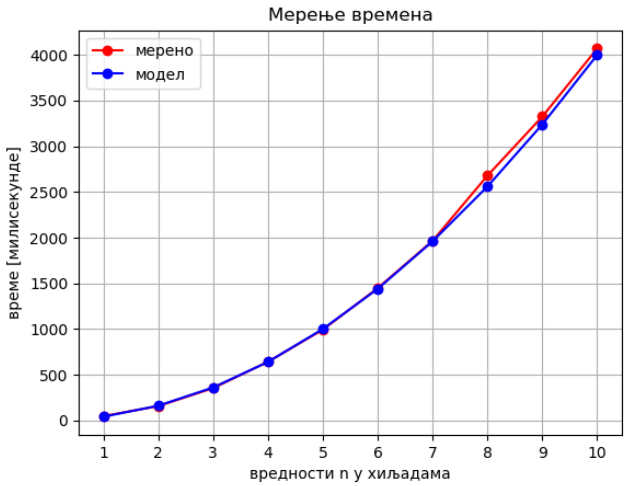


Слика 2.1: Линеарна зависност измереног времена

Ако се други алгоритам имплементира и ако се измери његово време извршавања (на пример, у програмском језику Python), добијају се следећи резултати (поново је приказан само збир свих бројева од 1 до n).

```
n =  1000, rezultat:   500500, vreme:    31.24 ms
n =  2000, rezultat:  2001000, vreme:   156.25 ms
n =  3000, rezultat:  4501500, vreme:   359.35 ms
n =  4000, rezultat:  8002000, vreme:   633.19 ms
n =  5000, rezultat: 12502500, vreme:   993.25 ms
n =  6000, rezultat: 18003000, vreme:  1448.38 ms
n =  7000, rezultat: 24503500, vreme:  1984.94 ms
n =  8000, rezultat: 32004000, vreme:  2547.53 ms
n =  9000, rezultat: 40504500, vreme:  3291.45 ms
n = 10000, rezultat: 50005000, vreme:  4049.11 ms
```

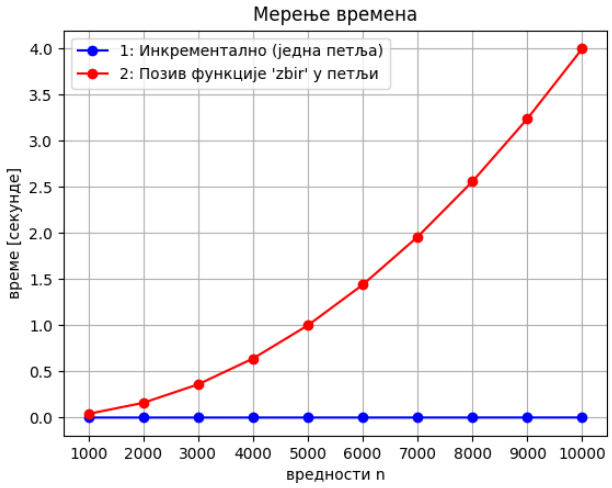
У овом случају зависност није линеарна. Наиме, када се n удвостручи, време се уместо 2 пута, отрпилике увећа 4 пута, што сугерише да је време извршавања сразмерно са квадратом величине улаза, тј. да је у питању квадратна зависност. Рачунањем $\frac{t(n)}{n^2}$ за разне n добијамо приближно сталну вредност 0.00004, што значи да се време извршавања $t(n)$ може апроксимирати као $t(n) \approx 0.00004n^2$. На следећој слици видимо график измерених вредности и график вредности добијене квадратне функције која моделира време извршавања.



Слика 2.2: Квадратна зависност измереног времена

Два дата графика се у значајној мери поклапају, што показује да је $0.00004n^2$ веома добра процена времена извршавања алгоритма за разне вредности n. Наравно, време извршвања увек зависи од програмског језика и конкретног рачунара на ком је мерење извршено, али експеримент говори да можемо очекивати да ће време рада другог алгоритма увек бити прилижно квадратна функција величине улаза. Ово се, наравно, може и формално доказати анализом броја извршених инструкција.

На основу измерених времена видимо да је први програм неупоредиво бржи у односу на други. Линеарно време извршавања је толико мање од квадратног, да када се оба времена прикажу на истом графику, делује да је линеарно време стално једнако нули.

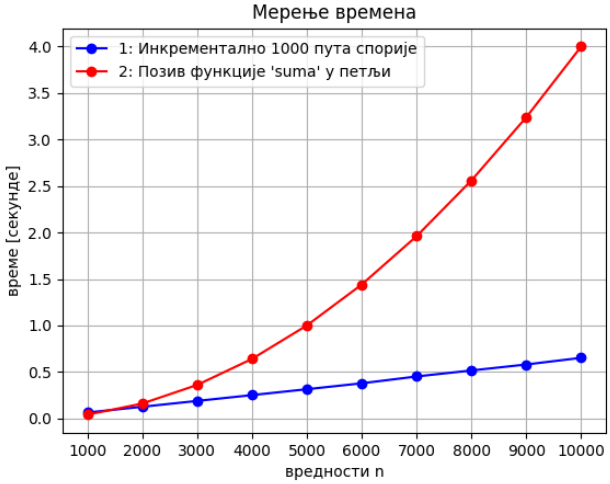


Слика 2.3: Однос линеарног и квадратног времена

2.2 Асимптотска анализа сложености

Мерење времена извршавања се може урадити за сваки програм, али оно није поуздан аргумент само за себе, већ служи пре свега за почетно стицање осећаја о ефикасности појединих алгоритама (да знамо шта треба да докажемо), као и за потврду закључака добијених теоријским разматрањем. Осим тога, често нам је потребно да можемо да унапред дамо неку грубу процену потребних ресурса за произвољне улазне вредности, без покретања програма (па чак и пре писања програма, само на основу алгоритма који ће бити примењен).

Питање које се природно поставља је то на ком ће се рачунару програм извршавати. Наравно, ако је један рачунар два пута бржи (у неком сегменту) од другог, за очекивати је да ће се програм на њему извршавати два пута брже. Ипак, показаће се да су разлике између ефикасних и неефикасних алгоритама толико велике, да је то што је неки рачунар 2, 3 или чак 10 пута бржи од другог заправо небитно и не може да надомести то колико је неефикасан алгоритам лошији од ефикасног. На пример, наредна слика приказује како би изгледао однос времена извршавања ако би се бржи алгоритам извршавао на 1000 пута споријем рачунару. Као што видимо, линеарна функција се мало "одлепила" од нуле, међутим, и даље су њене вредности много мање него код квадратне функције. Другим речима, алгоритам линеарне сложености, чак и када се успори 1000 пута, и даље је много бржи од алгоритма квадратне сложености (и што је n веће, разлика у брзини је све већа).



Слика 2.4: Однос линеарног и квадратног времена на рачунарима различите брзине

Да бисмо проценили зависност времена извршавања од димензије проблема, основни приступ је да покушамо да конструишемо функцију $f(n)$ која одређује зависност броја инструкција које алгоритам треба да изврши у односу на величину улаза n.

Израчунајмо број наредби сабирања које се изврше у првом и у другом програму из претходног примера (програми извршавају и друге наредбе, попут оних потребних да се организују петље, међутим, претпоставићемо да су нам сабирања једино значајна).

Јасно је да се у првом програму врши једно сабирање по петљи, па је укупан број сабирања једнак броју корака извршавања петље, а то је n. Ово је линеарна зависност, што је у складу са измереним временима.

У другом програму анализа је мало компликованија. За било које дато k, функција `zbir` врши око k сабирања. Пошто се функција позива за све вредности k од 1 до n, то се укупно изврши $1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$ сабирања. И ако бисмо рачунали све инструкције, добили бисмо да је број инструкција нека квадратна функција облика $an^2 + bn + c$, што је опет у складу са измереним временима.

Ако (поједностављено) претпоставимо да се свака инструкција на рачунару извршава за једну наносекунду ($10^{-9}s$), а да број инструкција зависи од величине улаза n на основу функције $f(n)$, тада је време потребно да се алгоритам изврши дат у следећим табелама.

Алгоритми чија је сложеност одозго ограничена полиномијалним функцијама, у принципу се сматрају ефикасним.

$n/f(n)$	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3
10	0,003 μs	0,003 μs	0,01 μs	0,033 μs	0,1 μs	1 μs
100	0,007 μs	0,010 μs	0,1 μs	0,644 μs	10 μs	1 ms
1,000	0,010 μs	0,032 μs	1,0 μs	9,966 μs	1 ms	1 s
10,000	0,013 μs	0,1 μs	10 μs	130 μs	0,1 s	16,7 min
100,000	0,017 μs	0,316 μs	100 μs	1,67 ms	10 s	11,57 dan
1,000,000	0,020 μs	1 μs	1 ms	19,93 ms	16,7 min	31,7 god
10,000,000	0,023 μs	3,16 μs	10 ms	0,23 s	1,16 dan	3×10^5 god
100,000,000	0,027 μs	10 μs	0,1 s	2,66 s	115,7 dan	
1,000,000,000	0,030 μs	31,62 μs	1 s	29,9 s	31,7 god	

Алгоритми чија је сложеност одоздо ограничена експоненцијалном или факторијелском функцијом се сматрају неефикасним.

$n/f(n)$	2^n	$n!$
10	1 μs	3,63 ms
20	1 ms	77,1 god
30	1 s	$8,4 \times 10^{15}$ god
40	18,3 min	
50	13 dan	
100	4×10^{13} god	

Можемо поставити и питање која димензија улаза се отприлике може обрадити за одређено време. Одговор је дат у наредној табели.

t	n	$n \log n$	n^2	n^3	2^n	$n!$
$1ms$	10^6	$63,000$	$1,000$	100	20	9
$10ms$	$10 \cdot 10^6$	$530,000$	$3,200$	215	23	10
$100ms$	$100 \cdot 10^6$	$4,5 \cdot 10^6$	$10,000$	465	27	11
$1s$	10^9	$40 \cdot 10^6$	$32,000$	$1,000$	30	12
$1min$	$60 \cdot 10^9$	$1,9 \cdot 10^9$	$245,000$	$3,900$	36	14

Из претходних табела јасно је да време извршавања суштински зависи од функције $f(n)$. На пример, ако поредимо алгоритме код којих је $f_1(n) = n$, $f_2(n) = 5n$, $f_3(n) = n^2$ и $f_4(n) = 2n^2 + 3n + 2$, јасно нам је да ће се за $n = 10^6$, време извршавања првог и другог алгоритма мерити милисекундама, док ће се време извршавања трећег и четвртог алгоритма мерити минутима. Код функције f_4, јасно је да је време које потиче од фактора $3n$ (три милисекунде) и 2 (две нанонсекунде) апсолутно занемариво у односу на време које долази од фактора $2n^2$ (око 33 минута). За прва два алгоритма рећи ћемо да имају *линеарну временску сложеност*, а за друга два да имају *квадратну временску сложеност*.

Чак ни педесет пута бржи рачунар неће помоћи да се трећи или четврти алгоритам изврше брже од првог или другог. Иако смо поједностављено претпоставили да се све инструкције извршавају исто време (једну наносекунду), што није случај у реалности, из претходних табела је јасно да нам тај поједностављени модел даје сасвим добру основу за поређење различитих алгоритама и да прецизнија анализа не би ни по чему значајно променила ситуацију. Сложеност се обично процењује на основу изворног кода програма. Савремени компилатори извршавају различите напредне оптимизације и машински кôд који се извршава може бити прилично другачији од изворног кода програма (на пример, компилатор може скупу операцију множења заменити ефикаснијим битовским операцијама, може наредбу која се више пута извршава у петљи изместити ван петље и слично). Детаљи који се у изворном коду не виде, попут питања да ли се неки податак налази у кеш-меморији или је потребно приступати РАМ-у, такође могу веома значајно да утичу на стварно време извршавања програма. Савремени процесори подржавају проточну обраду и паралелно извршавање инструкција, што такође чини стварно понашање програма другачијим од класичног, секвенцијалног модела који се најчешће подразумева приликом анализе алгоритама. Дакле, стварно време извршавања програма зависи од карактеристика конкретног рачунара на ком се програм извршава, али и од карактеристика програмског преводиоца, па и оперативног система на ком се програм извршава. Међутим, поново наглашавамо да ништа од тих фактора не може променити однос између времена извршавања алгоритама линеарне и алгоритама квадратне сложености, за велике улазе (код малих улаза, сви алгоритми раде веома ефикасно, па нам обрада малих улаза није интересантна).

Дакле, можемо закључити да нам за је за грубу процену времена потребног за извршавање неког алгоритма, чији број инструкција полиномијално зависи од величине улаза n довољно да знамо само који је степен тог полинома. Можемо слободно да занемаримо све мономе мањег степена, а можемо и слободно да занемаримо коефицијенте уз водећи степен, као и коефицијент којим се одређује брзина стварног рачунара у односу на овај фиктивни, за који смо приказали времена. Наиме у реалним ситуацијама сви ти коефицијенти могу да утичу да ће програм бити бржи или спорији највише десетак пута (па нек је и стотинак пута), али не могу да утичу на то да се за велики улаз алгоритам чији је број инструкција квадратни изврши брже од алгоритма чији је број инструкција линеаран (говоримо о односу минута и милисекунди).

Горња граница сложености се обично изражава коришћењем O-нотације.

Дефиниција: Ако постоје позитивна реална константа c и природан број n_0 такви да за функције f и g над

природним бројевима важи $f(n) \leq c \cdot g(n)$ за све природне бројеве n веће од n_0 онда пишемо $f(n) = O(g(n))$ и читамо "f је велико ‚о' од g".

У неким случајевима користимо и ознаку Θ која нам не даје само горњу границу, већ прецизно описује асимптотско понашање.

Дефиниција: Ако постоје позитивне реалне константе c_1 и c_2 и природан број n_0 такви да за функције f и g над природним бројевима важи $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ за све природне бројеве n веће од n_0, онда пишемо $f(n) = \Theta(g(n))$ и читамо „f је велико 'тета' од g".

Дакле, асимптотским ознакама смо занемарили мономе мањег степена и сакрили константе уз највећи степен полинома. Стварно време извршавања зависи и од константи сакривених у асимптотским ознакама, међутим, асимптотско понашање обично прилично добро одређује његов ред величине (да ли су у питању микросекунде, милисекунде, секунде, минути, сати, дани, године).

Наведимо карактеристике основних класа сложености.

- $O(1)$ – *константна сложеност*, алгоритми линијско-разгранате структуре који се извршавају практично моментално, нпр. алгоритми у којима се имплементира нека математичка формула;
- $O(\log n)$ – *логаритамска сложеност*, изузетно ефикасно, нпр. бинарна претрага;
- $O(\sqrt{n})$ – *коренска сложеност*, "логаритам за оне са јефтинијим улазницама" - немамо најбоља места, али ипак можемо да гледамо утакмицу, нпр. испитивање да ли је број прост, факторизација броја на просте чиниоце;
- $O(n)$ – линеарна сложеност, оптимално, када је за решење потребно погледати цео улаз, нпр. минимум/максимум серије елемената;
- $O(n \log n)$ – *квазилинеарна сложеност*, "линеарни алгоритам за оне са јефтинијим улазницама", ефикасни алгоритми засновани на декомпозицији (нпр. сортирање обједињавањем), ефикасном сортирању, коришћењу структура података са логаритамским временом приступа;
- $O(n^2)$ – *квадратна сложеност*, обично (али не обавезно) угнежђене петље, нпр. сортирање селекцијом, сортирање уметањем;
- $O(n^3)$ – *кубна сложеност*, обично (али не обавезно) вишеструко угнежђене петље, нпр. множење матрица;
- $O(2^n)$ – *експоненцијална сложеност*, изузетно неефикасно, нпр. испитивање свих подскупова;
- $O(n!)$ – *факторијелна сложеност*, изузетно неефикасно, нпр. испитивање свих пермутација.

Иако су класе поређане редом, не треба претпостављати да су оне "подједнако размакнуте". На пример, честа је заблуда да су алгоритми сложености $O(n \log n)$ по својој ефикасности негде између $O(n)$ и $O(n^2)$. Истина је заправо да су они прилично слични класи $O(n)$ и да је често тешко емпријски измерити разлику између те две класе, а да су и једни и други неупоредиво бржи од алгоритама квадратне сложености. На пример, ако је $n = 10^6$, веома груба процена (када се занемаре сви константни коефицијенти) броја корака линеарног алгоритма је 10^6, квазилинеарног алгоритма је око $20 \cdot 10^6$, а број корака квадратног алгоритма је 10^{12}. Дакле, на тако великом улазу квазилинеарни алгоритам би био тек пар десетина пута спорији од линеарног (конкретан однос, наравно, зависио би од константих фактора), док би квадратни алгоритам био милион пута спорији од линеарног и неких педесет хиљада пута спорији од квазилинеарног.

2.3 Сложеност неких честих облика петљи

Прикажимо сада кроз неколико примера анализу сложености итеративно имплементираних алгоритама. Иако нећемо посматрати решења конкретних задатака, потрудићемо се да у примерима покријемо облике петљи који се јављају у великом броју конкретних алгоритама и решења конкретних задатака.

У примерима петљи који следе, претпоставља се да кôд у телу петљи који није приказан не утиче на бројачке променљиве и не мења границе петљи.

```
for (int i = 0; i < n; i++)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n)$.

```
for (int i = m; i < n; i++)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n - m)$.

```
for (int i = 0; i < n; i += 2)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n)$. Пошто се петља извршава за парне вредности бројачке променљиве, тело петље се извршава око $\frac{n}{2}$ пута и константни фактор је $\frac{1}{2}$, али је сложеност и даље линеарна.

```
for (int i = 0, j = n-1; i < j; i++, j--)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n)$. Показивачи се сусрећу приближно на средини опсега, а тело петље се извршава око $\frac{n}{2}$ пута.

```
for (int i = 0; i < m; i++)
   for (int j = 0; j < n; j++)
      // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(mn)$. Заиста, спољашња петља се извршава m, а у њеном телу се унутрашња петља извршава n пута, па се тело унутрашње петље изврши тачно mn пута.

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^2)$. Заиста, спољашња петља се извршава n, а у њеном телу се унутрашња петља извршава n пута, па се тело унутрашње петље изврши тачно n^2 пута.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^2)$. Број извршавања тела унутрашње петље је $(n-1)+(n-2)+\ldots+2+1$, што је једнако $\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$. Константни фактор је $\frac{1}{2}$, али је сложеност квадратна. До истог резултата можемо доћи ако схватимо да у сваком кораку унутрашње петље пар бројача одређује једну комбинацију бројева од 0 до $n-1$. Зато број извршавања тела одговара броју двочланих комбинација скупа од n елемената, што је једнако $\binom{n}{2} = \frac{n(n-1)}{2}$.

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < n; j++)
         // kod slozenosti O(1)
```

Сложеност претходних петљи је прилично очигледно $O(n^3)$.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      for (int k = j+1; k < n; j++)
         // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^3)$. Најлакши начин да се ово закључи је да се примети да сваком извршавању тела одговара једна трочлана комбинација елемената скупа $0, \ldots, n-1$. Пошто трочланих комбинација има $\binom{n}{3} = \frac{n(n-1)(n-2)}{3 \cdot 2 \cdot 1}$, сложеност је кубна, а константни фактор је $\frac{1}{6}$.

```
for (int i = 0; i < m; i++)
    // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(m+n)$. Наиме, тело прве петља се изврши m пута, а затим тело друге петље n пута, па се тела обе петље укупно изврше $m+n$ пута.

```
for (int i = 0; i < n; i++)
    // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n)$. Наиме, тело прве петље се изврши n пута, а затим тело друге n пута, па се тела обе петље укупно изврше $2n$ пута, но то је $O(n)$.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
       // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^2)$. Наиме, тело угнежђених петљи се изврши $\frac{n(n+1)}{2}$ пута, а затим тело друге петље n пута, што је заправо занемариво мало у односу на број извршавања тела угнежђених петљи. Дакле, први део кода апсолутно доминира временом извршавања и сложеност је $O(n^2)$.

Могло би се помислити да број угнежђених петљи одговара степену полинома, али то није увек случај.

```
for (int i = 1; i*i <= n; i++)
   // kod slozenosti O(1)
```

Иако садржи једну петљу, сложеност претходног кода није $O(n)$, већ $O(\sqrt{n})$. Наиме, петља се извршава све док је $i^2 \leq n$ тј. док је $i \leq \sqrt{n}$.

```
for (int i = 1; i < n; i *= 2)
   // kod slozenosti O(1)
```

Иако претходни код садржи петљу, његова сложеност је $O(\log n)$, јер се вредност промењиве `i` дуплира у сваком кораку, све док не престигне граничну вредност n.

```
for (int i = 0; i < 10; i++)
   // kod slozenosti O(1)
```

Сложеност претходног кода је $O(1)$. Иако је присутна петља, број њених извршавања је увек 10 и не зависи ни од једног параметара, па га можемо сматрати малом константом.

```
for (int i = 1; i < n; i++)
    for (int j = 1; j < i; j *= 2)
          // kod slozenosti O(1)
```

Сложеност претходног кода је $O(n \log n)$. Сложеност унутрашње петље, за свако конкретно `i` је $O(\log i)$, па је укупна сложеност отприлике једнака $\log 1 + \log 2 + \ldots + \log n$, а за ово се може показати да је $O(n \log n)$ (јасно је да је израз мањи или једнак $n \log n$ јер је сваки сабирак мањи или једнак $\log n$, међутим, може се показати и да је збир већи или једнак од $\frac{n}{2} \log \frac{n}{2}$ (што је такође $\Theta(n \log n)$), занемаривањем првих $\frac{n}{2}$ сабирака, након чега остају само сабирци који су већи или једнаки $\log \frac{n}{2}$).

```
for (int i = n; i >= 1; i /= 2)
    for (int j = 1; j < i; j++)
          // kod slozenosti O(1)
```

Сложеност претходног кода је $O(n)$. Наиме, број извршавања унутрашње петље је $n + \frac{n}{2} + \frac{n}{4} + \ldots$, за шта се лако може показати да је одозго ограничено са $2n$.

```
for (int i = 0; i < n; i++) {
   for (int j = i+1; j < n && P(a[j]); j++)
          // kod slozenosti O(1)
   // kod slozenosti O(1)
   i = j;
}
```

Овакав код се може срести у алгоритмима у којима се анализирају све серије узастопних елемената низа који задовољавају неко својство `P` (на пример, коришћењем петљи овог облика можемо пронаћи најдужу серију узастопних парних елемената).

Иако постоје угнежђене петље, сложеност претходног кода је $O(n)$. Број извршавања унутрашње петље зависи од стања низа `a`, па не знамо унапред ни колико пута ће та петља бити покренута нити колико ће се пута њено тело извршити при сваком покретању. Међутим, да бисмо одредили укупну сложеност целог кода, то нам није ни потребно. Можемо извршити амортизовану анализу и израчунати укупан број извршавања тела

унутрашње петље. Кључни детаљ је то што унутрашња петља креће од текуће вредности спољашње бројачке променљиве, док спољашња бројачка променљива након унутрашње петље наставља тамо где се унутрашња петља завршила. Зато при сваком пролазу кроз тело унутрашње петље променљива `j` има строго вредност већу него при сваком претходном пролазу, што се може десити највише n пута.

```
int j = 0;
for (int i = 0; i < n; i++) {
    while (j < n && P[j]) {
       // kod slozenosti O(1)
       j++;
    }
   // kod slozenosti O(1)
}
```

Иако постоје угнежђене петље, сложеност претходне петље је $O(n)$. Кључни детаљ је то што унутрашња петља нема иницијализацију променљиве `j` на нулу и бројач `j` у унутрашњој петљи се све време само увећава (исто као и бројач `i` у спољашњој петљи). Укупан број корака је стога ограничен са $2n$.

```
int l = 0, d = n-1;
while (l < d) {
  do l++; while (l < d && P(a[l]));
  do d--; while (l < d && Q(a[d]));
  if (l < d)
     // kod slozenosti O(1)
}
```

Сличан кôд се, на пример, може срести у алгоритму партиционисања низа. И претходни алгоритам је сложености $O(n)$ иако и он садржи угнежђене петље. То поново можемо утврдити амортизованом анализом (јер не знамо појединачни број извршавања унутрашњих петљи, али можемо лако проценити укупан корака који се у њима направи). Наиме, променљива `l` се само увећава кренувши од почетка, а `d` се само смањује кренувши од краја низа, док се не сусретну, што ће се десити у највише n корака.

Дакле, иако нам у већини случајева угнежђеност петљи описује сложеност алгоритма, треба бити обазрив и код анализирати пажљивије, да се сложеност не би преценила.

2.4 Скривена сложеност

Често процену сложености грубо вршимо тако што анализирамо структуру петљи у програму, занемарујући остале операције (обично за све сем петљи сматрамо да је $O(1)$). То може бити прилично варљиво, јер се у коду могу позивати функције (било кориснички дефинисане, било библиотечке) које нису константне сложености. Још горе, и неки оператори могу бити неконстантне сложености (обично линеарне).

```
string rez = "";
foreach (char c : s)
   rez = c + rez;
```

Иако има само једну петљу, претходни фрагмент може бити сложености $O(n^2)$, где је n дужина ниске `s`. Наиме, додавање карактера на почетак ниске у језику C++ може бити линеарне сложености $O(n)$, где је n дужина ниске (јер захтева померање наредних карактера надесно).

2.5 Савети за побољшање сложености

Кључни савет за побољшање сложености је то да рачунар ради само оно што је неопходно да би се добио коначан резултат. Када се та идеја мало детаљније разради, добијамо следећи низ савета који нас често доводе до алгоритама мање сложености:

- Немој терати рачунар да врши дуготрајна израчунавања која се могу извршити и "пешке", применом математике.
- Немој терати рачунар да више пута израчунава једно те исто – упамти потребне резултате израчунавања у меморији, да их не би рачунао више пута.
- Немој терати рачунар да израчунава ствари које нису потребне за добијање коначног решења проблема.

- Немој терати рачунар да испитује случајеве за које унапред можеш закључити да не могу бити тражено решење проблема.
- Ако је то могуће, припреми податке тако да се касније могу ефикасније обрадити.
- Користи ефикасније структуре података.
- …

У наставку овог поглавља приказаћемо низ задатака које ћемо решити различитим алгоритмима, анализираћемо њихову асимптотску сложеност најгорег случаја и приказаћемо како се на бази приказаних савета могу изградити значајно ефикаснији алгоритми.

2.6 Замена итерације формулом

Један од важних савета за побољшање сложености алгоритама је тај да не терамо рачунар да врши дуготрајна израчунавања која се могу извршити и "пешке", применом математике. Наиме, често се одређене вредности израчунавају применом итеративних поступака. На пример, збир елемената неког низа израчунавамо додавањем једног по једног елемента. То даје тражени резултат, али и одузима неко време. Постоје ситуације када су елементи који се обрађују правилни и када се коначан резултат може добити применом неке задате формуле, без примене итеративног поступка. На пример, ако знамо да треба сабрати првих n елемената низа $1, 2, 3, \ldots, n$, нема потребе да примењујемо итеративни поступак сабирања чија је сложеност $O(n)$, већ је довољно да у времену $O(1)$ применимо Гаусову формулу на основу које знамо да је

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

Још неке формуле се често могу употребити за смањење сложености (али и за саму анализу сложености), па ћемо их у наставку навести.

2.6.1 Аритметички и геометријски низ

Сличне формуле које су нам корисне су формуле за n-ти члан и збир првих n елемената аритметичког низа a_0, $a_0 + d$, $a_0 + 2d$, ….

$$a_i = a_0 + (i-1)d, \qquad \sum_{i=0}^{n} a_i = \frac{n(a_0 + a_n)}{2},$$

као и за n-ти члан и збир првих n елемената геометријског низа a_0, $_0 \cdot q$, $a_0 \cdot q^2$, …

$$a_i = a_0 \cdot q^i, \qquad \sum_{i=0}^{n} a_i = a_0 \frac{1 - q^{n+1}}{1 - q}.$$

2.6.2 Збирови степена

$$1^2 + 2^2 + \ldots + n^2 = \sum_{k=1}^{n} k^2 = \frac{n \cdot (n + \frac{1}{2}) \cdot (n+1)}{3}$$

$$1^3 + 2^3 + \ldots + n^3 = \sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2.$$

2.6.3 Комбинаторика

- Број начина да се из скупа од n различитих бројева извуку два броја $a < b$ је $\binom{n}{2} = \frac{n(n-1)}{2}$.
- Број начина да се из скупа од n различитих бројева извуку три броја $a < b < c$ је $\binom{n}{3} = \frac{n(n-1)(n-2)}{3 \cdot 2}$.

Задатак: Спортске припреме

Ватерполистима клуба "Делфини" организоване су припреме у трајању од n дана. Првог дана припрема сваки ватерполисти преплива по a метара, а сваког следећег дана за d метара више него претходног дана. Написати програм којим се одређује колико укупно километара, током тих припрема, преплива сваки ватерполиста клуба "Делфини".

Улаз: Први ред стандардног улаза садржи број дана припрема n ($n \leq 10^5$). Сваки наредни ред стандардног улаза представља податке за једног ватерполисту (њих највише 10^5). Уносе се по два броја раздвојена размаком: број метара који ватерполиста преплива првог дана припрема a ($a \leq 3000$), и број метара које сваки дан више плива у односу на претходни дан d ($d \leq 1000$).

Излаз: На стандардном излазу за сваког ватерполисту приказати колико километара укупно преплива током припрема.

Пример

Улаз	*Излаз*
`10`	`9.50`
`500 100`	`4.9`
`400 20`	

Решење

Итеративно сабирање

Сабирање је могуће реализовати и коришћењем петље (алгоритмом сабирања серије бројева, слично као у задатку Збир n бројева), али тако добијамо прилично неефикасан програм.

Време израчунавања за сваког ватерполисту било би линеарно ($O(n)$), док се за m ватерполиста резултати израчунавају у времену $O(mn)$.

Пошто се у овом задатку тесно преплићу фаза учитавања података и исписа резултата, пожељно је програм оптимизовати аутоматском тестирању (помоћу `cin.tie(0)` и коришћењем `\n` уместо `endl`).

```
#include <iostream>
#include<iomanip>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n, a, d;
  cin >> n;
  while (cin >> a >> d) {
    // koliko je presao ukupno
    long long S = a;
    // koliko je presao u tekucem danu
    int x = a;
    for (int i = 1; i < n; i++) {
      x += d;
      S += x;
    }
    double Skm = S / 1000.0;
    cout << fixed << setprecision(2) << showpoint << Skm << '\n';
  }
  return 0;
}
```

Формула за збир елемената аритметичког низа

Дневне вредности које сваки ватерполиста преплива представљају аритметички низ и тражена вредност је сума првих n чланова тог низа.

Обележимо са a_i колико метара је сваки ватерполиста клуба "Делфини" препливао i-тог дана припрема ($1 \leq i \leq n$). Једноставном анализом (погледати задатак N-ти дан тренинга) можемо закључити да је $a_i = a + (i -$

$1) \cdot d$. Наш задатак је да одредимо суму $a_1 + a_2 + ... + a_{n-1} + a_n$. Збир првог и последњег члана, једнак је збиру другог и претпоследњег, трећег и претпретпоследњег итд. Постоји дакле $\frac{n}{2}$ сабирака једнаких, $a_1 + a_n$, па је тражени збир једнак

$$\frac{n(a_1 + a_n)}{2}.$$

Формално, обележимо тражену суму са S. Применом формуле за a_i добијамо да је

$$S = a + (a + d) + ... + (a + (n - 2) \cdot d) + (a + (n - 1) \cdot d)$$

Ако тражену суму испишемо и у обратном поретку добијамо:

$$S = (a + (n - 1) \cdot d) + (a + (n - 2) \cdot d) + ... + (a + d) + a$$

Сабирањем последње две једнакости члан по члан (сабирамо прве чланове, друге чланове, итд.) добијамо:

$$2 \cdot S = (2 \cdot a + (n - 1) \cdot d) + (2 \cdot a + (n - 1) \cdot d) + ... + (2 \cdot a + (n - 1) \cdot d) + (2 \cdot a + (n - 1) \cdot d)$$

односно $2 \cdot S = n \cdot (2 \cdot a + (n - 1) \cdot d)$. Из последње једнакости добијамо

$$S = \frac{n \cdot (2 \cdot a + (n - 1) \cdot d)}{2} = na + \frac{n(n - 1)}{2},$$

што је тражена коначна формула.

Приметимо да је производ $n \cdot (2 \cdot a + (n - 1) \cdot d)$ дељив са 2 јер је $2 \cdot a$ парно и паран је број n или $n - 1$, па је према томе један од фактора посматраног производа паран и могуће је употребити целобројно дељење бројем 2.

На овај начин добијамо укупан број метара које је сваки ватерполиста препливао у току припрема. Да би добили број препливаних километара потребно је добијену суму поделити са 1000 (тачније са 1000.0 да би резултат био реалан број).

Да не би дошло до прекорачења приликом множења, потребно је податке регистровати 64-битним целобројним типом.

Време израчунавања за сваког ватерполисту је константно ($O(1)$), док се за m ватерполиста резултати израчунавају у времену $O(m)$.

```
#include <iostream>
#include<iomanip>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n, a, d;
  cin >> n;
  while (cin >> a >> d) {
    long long S = (long long) n * (long long) (2*a + (n-1)*d) / 2;
    double Skm = S / 1000.0;
    cout << fixed << setprecision(2) << showpoint << Skm << '\n';
  }
  return 0;
}
```

Гаусова формула

Други начин да дођемо до резултата је да приметимо да је $S = n \cdot a + d \cdot (1 + 2 + \ldots + n - 1)$, и да се позовемо на Гаусову формулу која каже да за суму природних бројева од 1 до n важи

$$1 + 2 + \ldots + (n-1) + n = \frac{n(n+1)}{2}.$$

Зато је сума бројева од 1 до $n-1$ једнака $\frac{(n-1)n}{2}$.

Једна анегдота каже да је Гаус још као дете израчунао збир бројева од 1 до 100 тако што је приметио да се међу њима налази 50 парова бројева чији је збир 101, тако да се одређивање збира првих природних бројева на тај начин приписује Гаусу.

```
#include <iostream>
#include <iomanip>

using namespace std;

// zbir brojeva 1, 2, ..., n
long long gaus(long long n) {
  return  n * (n+1) / 2;
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n, a, d;
  cin >> n;
  while (cin >> a >> d) {
    long long S = n*a + d*gaus(n-1);
    double Skm = S / 1000.0;
    cout << fixed << setprecision(2) << showpoint << Skm << '\n';
  }
  return 0;
}
```

Задатак: N-ти дан тренинга

На планети супер-хероја, ватерполисти учествују на припремама за такмичење у трајању од n дана. Првог дана припрема ватерполиста преплива a метара, а сваког следећег дана за d метара више него претходног дана. Написати програм којим се за сваког ватерполисту у клубу одређује колико метара преплива последњег дана припрема.

Улаз: Први ред стандардног улаза садржи број дана припрема n ($n \leq 10^5$). Сваки наредни ред стандардног улаза представља податке за једног ватерполисту (њих највише 10^5). Уносе се три два броја раздвојена размаком: број метара који ватерполиста преплива првог дана припрема a ($a \leq 3000$), и број метара које сваки дан више плива у односу на претходни дан d ($d \leq 1000$).

Излаз: На стандардном излазу за сваког ватерполисту приказати колико метара преплива последњег дана припрема.

Пример

Улаз	*Излаз*
10	1400
500 100	580
400 20	

Задатак: Аритметички троугао

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Итерација

До решења се може доћи коришћењем петљи. У првој петљи одређујемо први елемент k-тог реда, а уз то одређујемо и број елемената у њему. Први елемент одређујемо тако што саберемо број елемената у претходних $k-1$ редова, док број елемената сваког реда тако што у сваком кораку број елемената претходног реда увећавамо за два (сваки наредни ред има тачно два елемента више од претходног). Дакле, у петљи одржавамо почетак и број елемената текућег реда (иницијализујемо их на један, јер први ред почиње од један и има тачно један елемент) и $k-1$ пута почетак реда увећавамо за број елемената текућег реда, а број елемената текућег реда за два.

Прикажимо ово на примеру одређивања првог елемента реда 5.

k	pocetak	brojElemenata
1	1	1
2	2	3
3	5	5
4	10	7
5	17	9

Након тога, у другој петљи одређујемо збир елемената у k-том реду, тако што на збир који иницијализујемо на нулу додајемо један по један елемент тог реда, коришћењем алгоритма сабирања серије бројева – елементи су узастопни природни бројеви, па их је лако набројати.

У првој петљи обилазимо k редова којима одређујемо почетак помоћу две операције сабирања. Пажљивом анализом можемо закључити да у реду k има $2k-1$ елемената (мада ту чињеницу нисмо употребили у програму), који се у другој фази сабирају. Сложеност обе фазе, па и укупног решења је, дакле, $O(k)$. Задатак, наравно, може да се реши и у мањој сложености од ове, без коришћења петљи.

Докажимо формално коректност овог поступка. Инваријанта петље је да након m извршавања њеног тела важи да је $i = m+1$ и $1 \leq i \leq k$, као и да променљива `pocetak` садржи први елемент реда $i = m+1$, а променљива `brojElemenata` садржи број елемената реда $i = m+1$.

- Пошто се променљива `i` иницијализује на вредност 1, након $m = 0$ корака важи да је $1 = 0+1$ и $1 \leq 1 \leq k$. Пошто су и обе променљиве иницијализоване на вредност 1, након $m = 0$ корака петље, променљиве заиста садрже први елеменат и број елемената реда $i = m+1 = 0+1 = 1$.
- Претпоставимо да инваријанта важи након m корака петље и да је услов петље испуњен тј. да је $i < k$.

  Докажимо да тврђење важи и након $m' = m+1$ извршавања тела и корака петље. Пошто након извршавања корака важи $i' = i+1$, а пошто је на основу индуктивне хипотезе важило да је $i = m+1$, важи и да је $i' = m'+1$. Пошто је услов петље био испуњен, важило је $i < k$, па уз индуктивну претпоставку $1 \leq i \leq k$, важи $1 \leq i' \leq k$.

  Означимо са p и b вредности променљивих `pocetak` и `brojElemenata` на улазу у тело петље, а са p' и b' њихове нове вредности, након извршења тела и корака петље. Анализом додела у телу петље, јасно видимо да је $p' = p+b$ и $b' = b+2$. На основу претпоставке важи да је p први елемент реда $i = m+1$, а да је b број елемената реда $i = m+1$. Сабирањем првог елемента било ког реда у троугла и броја елемената тог реда троугла добија се први елемент наредног реда троугла. Дакле, важи да је $p+b$ први елемент реда $i+1 = m+2$ троугла, па, важи да је $p' = p+b$ први елемент реда $i' = i+1 = m+2 = m'+1$. На основу дефиниције троугла важи јасно и да сваки наредни ред има и два елемента више него претходни. Зато, пошто је b број елемената реда $m+1$, важи и да је $b' = b+2$ број елемената реда $i' = m'+1$.
- На крају петље услов није испуњен, па важи да је $i \geq k$. Уз услов $1 \leq i \leq k$, мора да важи да је $i = k$. На основу инваријанте знамо да променљива `pocetak` садржи број елемената реда $i = m+1 = k$. Дакле, петља је коректно извршила свој задатак и у променљиву `pocetak` сместила први елемент реда k.

На сличан начин се може формално доказати и коректност друге петље (инваријанта је да након m њених корака променљива `zbirRedaTrougla` садржи збир првих m елемената реда k, а да променљива i садржи вредност елемента на позицији m у том реду, ако се позиције броје од 0).

Приметимо да смо у овој петљи увели и нову променљиву којом смо регистровали број елемената у текућем реду троугла. То што смо ту променљиву имали на располагању нам је помогло да ажурирамо вредност почетног елемента наредног реда, међутим, "кредит" који смо добили на почетку тела петље морали смо да вратимо на крају тако што смо морали да ажурирамо и вредност те променљиве (и тако је припремимо

за наредну итерацију). Ова техника је позната под именом **ојачавање индуктивне хипотезе** и често се користи приликом конструкције алгоритама.

```
#include <iostream>
using namespace std;

int main() {
  long long k;
  cin >> k;

  // odredjujemo prvi broj u k-tom redu trougla
  long long pocetak = 1;
  long long brojElemenata = 1;
  for (int i = 1; i < k; i++) {
    pocetak += brojElemenata;
    brojElemenata += 2;
  }

  // odredjujemo zbir elemenata u k-tom redu trougla
  long long zbirRedaTrougla = 0;
  for (long long i = pocetak; i < pocetak + brojElemenata; i++)
    zbirRedaTrougla += i;

  cout << zbirRedaTrougla << endl;
  return 0;
}
```

Формула за збир аритметичког низа

Приметимо да последњи елемент у сваком реду троугла представља потпун квадрат (то су бројеви 1, 4, 9, 16, итд). Дакле, пошто бројање редова почиње од 1, последњи елемент у реду број k је број k^2. Зато је први број у реду број k једнак $(k-1)^2+1$ (следбеник последњег броја у претходном реду, који је потпун квадрат). У реду број k има $n = 2k-1$ елемент и чланови реда су узастопни природни бројеви, тако да чине аритметички низ (види задатак Спортске припреме), чији је први члан једнак $a_1 = (k-1)^2+1$, а разлика $d = 1$. Тражени збир можемо израчунати на основу формуле за збир елемената аритметичког низа $n \cdot a_1 + d\frac{(n-1)n}{2}$. Уврштавањем вредности добијамо

$$(2k-1)((k-1)^2+1) + \frac{(2k-2)(2k-1)}{2} =$$

$$(2k-1)\left((k-1)^2+1+(k-1)\right) =$$

$$(2k-1)(k^2-k+1)$$

Нагласимо још и да број елемената у сваком реду троугла чини аритметички низ са првим чланом 1 и разликом 2 (у сваком реду има два реда више него у претходном). То је аритметички низ који чине сви непарни бројеви, његов k-ти члан је $1+2(k-1) = 2k-1$ (види задатак N-ти дан тренинга), док је збир његових k елемената једнак $k+2\frac{k(k-1)}{2} = k^2$, што заправо доказује да у реду k има $2k-1$ елемената, а да је последњи елемент у том реду заиста k^2.

Применом изведених формула, решење из израчунава у сложености $O(1)$.

```
#include <iostream>
using namespace std;

int main() {
  long long k;
  cin >> k;
```

```
  long long zbirRedaTrougla = (2 * k - 1) * (k * k - k + 1);
  cout << zbirRedaTrougla << endl;
  return 0;
}
```

Читљивости програма ради, уместо финалне формуле можемо употребити функцију за израчунавање збира елемената аритметичког низа.

```
#include <iostream>
using namespace std;

// zbir niza a1, a1+d, ..., a1+(n-1)d
long long zbirAritmetickogNiza(long long a1, long long d, long long n) {
  return n * a1 + d * (n-1) * n / 2;
}

int main() {
  long long k;
  cin >> k;
  long long a1 = (k - 1) * (k - 1) + 1;
  long long d = 1;
  long long n = 2 * k - 1;
  long long zbirRedaTrougla = zbirAritmetickogNiza(a1, d, n);
  cout << zbirRedaTrougla << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Аритметички квадрат

Бројеви од 0 до $n^2 - 1$ређају се у квадрат. На пример, за $n = 5$ добија се следећи квадрат:

```
0   1  2  3  4
5   6  7  8  9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

Напиши програм који израчунава збир бројева у датој врсти и датој колони квадрата (и њих ћемо бројати од нуле).

Улаз: Са стандардног улаза учитавају се следећи цели бројеви (сваки у посебном реду).

- n - димензија квадрата ($1 < n \leq 10^9$)
- i - број врсте и колоне чији се збир тражи ($0 \leq i < n$)

Излаз: На стандардни излаз исписати два цела броја, сваки у посебном реду:

- збир бројева у i-тој врсти квадрата и
- збир бројева у i-тој колони квадрата

Ако су ови збирови већи или једнаки од 10^9, исписати њихових последњих 9 цифара.

Пример

Улаз	*Излаз*
5	35
1	55

Задатак: Број подстрингова који почињу и завршавају са 1

Дат је бинарни стринг (ниска карактера која се састоји од карактера 0 и 1). Написати програм којим се одређује број сегмената (подстринг узастопних елемената), дужине најмање 2, који почињу и завршавају са 1.

Улаз: Прва и једина линија стандардног улаза садржи бинарни стринг (састављен од 0 и 1).

Излаз: На стандардном излазу приказати у једној линији тражени број сегмената.

Пример

Улаз	*Излаз*
`010001001`	`3`

Објашњење

То су подстрингови `10001`, `10001001` и `1001`.

Решење

Анализа свих сегмената

Број свих сегмената који почињу и завршавају са 1 можемо једноставно одредити анализирајући све сегменте. У спољашњој петљи анализирамо један по један карактер. Сваку јединицу на коју наиђемо (за свако i такво да је s_i једнако 1), разматрамо као почетак сегмента и у унутрашњој петљи (бројачем j од $i + 1$ до краја стринга) тражимо јединицу којом се сегмент завршава. За сваку јединицу пронађену у унутрашњој петљи (за свако j такво да је s_j једнако 1) увећавамо број сегмената.

Приметимо да на овај начин исте карактере стринга непотребно анализирамо велики број пута. Сложеност алгоритма одговара укупном броју свих сегмената и једнака је $O(n^2)$.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string s;
  cin >> s;
  int n = s.length();
  int br = 0;
  for (int i = 0; i < n - 1; i++)
    if (s[i] == '1')
      for (int j = i + 1; j < n; j++)
        if (s[j] == '1')
          br++;
  cout << br << endl;
  return 0;
}
```

Бројање јединица

Сваки сегмент који почиње и који се заврашава јединицом дефинисан је позицијама две јединице у стрингу, па је укупан број тражених сегмената једнак броју начина да се изаберу две различите јединице у стрингу. Ако је укупан број јединица у стрингу b онда две јединице можемо изабрати на $\frac{b \cdot (b-1)}{2}$ начина.

Пошто јединице можемо пребројати само једним проласком кроз стринг, сложеност овог алгоритма је $O(n)$.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string s;
  cin >> s;

  int brojJedinica = 0;
  for (char c : s)
```

```
    if (c == '1')
      brojJedinica++;

  cout << brojJedinica * (brojJedinica - 1) / 2 << endl;

  return 0;
}
```

Задатак: Недостајући број

У низу бројева од 0 до n тачно један број је изостављен. Напиши програм који, без памћења елемената низа, учитава бројеве са улаза и ефикасно одређује који број недостаје.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^9$), а затим и описани низ бројева (бројеви су наведени у једном реду, раздвојени са по једним размаком).

Излаз: На стандардни излаз исписати елемент који недостаје.

Пример

Улаз	*Излаз*
`5` `0 4 2 5 1`	`3`

Решење

Линеарна претрага

Једно решење може бити засновано на линеарној претрази свих кандидата. Оно подразумева да су сви елементи учитани у низ (што, додуше, нарушава услов који је дат у поставци задатка). За све бројеве од 0 до n проверавамо да ли су садржани у низу.

У језику C++ линеарну претрагу можемо реализовати библиотечком функцијом `find` којој се прослеђују итератори на део низа који се претражује и вредност која се тражи. Функција ће вратити итератор на пронађено прво појављивање елемента или итератор који указује непосредно иза краја претраженог распона, ако елемент не постоји.

Пошто су елементи учитани у низ дужине n, меморијска сложеност је $O(n)$.

Линеарна претрага низа од n елемената у најгорем случају захтева $O(n)$ корака, па пошто се тражи $n + 1$ елемент, сложеност је $O(n^2)$. Може се приметити и да није могуће да се у сваком кораку линеарне претраге догоди најгори случај. Најгори случај се дешава када се елемент не налази у низу, а сви елементи који се траже (сем једног) су присутни. Заиста, када тражимо елемент који се налази на позицији i та претрага ће се завршити у i корака. Прецизније, елемент на позицији 1 ће се пронаћи у једном кораку, елемент на позицији 2 у два корака, елемент на позицији 3 у три корака и тако даље. Укупан број корака је зато $1 + 2 + \ldots + n$, што је опет $O(n^2)$.

Разним техникама временска сложеност се може свести на $O(n)$, а циљ нам је и да меморијску сложеност сведемо на $O(1)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  for (int x = 0; x <= n; x++)
```

```
    if (find(begin(a), end(a), x) == end(a)) {
      cout << x << endl;
      break;
    }
  return 0;
}
```

Збир елемената

Збир свих елемената из скупа $\{0, 1, 2, \ldots, n\}$ је $0 + 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$. То је збир елемената који се налазе у низу и недостајућег елемента. Недостајући елемент је, дакле, једнак, разлици између $\frac{n(n+1)}{2}$ и збира свих елемената низа. Потребно је обратити пажњу на то да је за израчунавање збира елемената потребно користити бар 64-битни целобројни тип.

Збир свих елемената лако можемо израчунати у времену $O(n)$, приликом учитавања (без смештања елемената у низ, па је меморијска сложеност $O(1)$).

```
#include <iostream>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  long long zbir = 0;
  for (int i = 0; i < n; i++) {
    int x; cin >> x;
    zbir += x;
  }
  long long zbir_svih = ((long long)n) * (n+1) / 2;
  int nedostajuci = zbir_svih - zbir;
  cout << nedostajuci << endl;
  return 0;
}
```

Задатак: Број дељивих у интервалу

Напиши програм који одређује колико у интервалу $[a, b]$ постоји бројева дељивих бројем k.

Улаз: Са стандардног улаза уносе се три цела броја, сваки у посебном реду.

- a ($0 \leq a \leq 10^9$)
- b ($a \leq b \leq 10^9$)
- k ($1 \leq k \leq 10^9$)

Излаз: На стандардни излаз исписати тражени цео број.

Пример

Улаз	*Излаз*
30 53 5	5

Објашњење

Бројеви су 30, 35, 40, 45 и 50.

Решење

Линеарна претрага

Могуће је направити решење засновано на претрази грубом силом, које укључује коришћење петљи. То решење је најједноставније за разумевање и имплементацију, међутим може бити неефикасно за велику разлику између бројева a и b. Потребно је у петљи, редом проћи кроз све бројеве од a до b, проверити за сваки да ли је дељив бројем k и за сваки који јесте, увећати бројач пронађених бројева. Дакле, овај алгоритам врши бројање елемената филтриране серије узастопних природних бројева из интервала $[a, b]$, а филтрирање се врши на основу провере дељивости која се своди на поређење остатака при дељењу са нулом.

Сложеност овог решења је линеарна у односу на број елемената у интревалу тј. $O(b - a)$.

```
#include <iostream>

using namespace std;

int main() {
  int a, b, k;
  cin >> a >> b >> k;
  int broj = 0;
  for (int i = a; i <= b; i++)
    if (i % k == 0)
      broj++;
  cout << broj << endl;
  return 0;
}
```

Израчунавање броја

Да би број x био дељив бројем k потребно је да постоји неко n тако да је $x = n \cdot k$. Пошто x мора бити у интервалу $[a, b]$, мора да важи да је $a \leq n \cdot k$ и $n \cdot k \leq b$. Из задатака Поклони и Лифт знамо да је најмање n које задовољава прву неједначину једнако $n_l = \lceil \frac{a}{k} \rceil$, а да је највеће n које задовољава другу неједначину једнако $n_d = \lfloor \frac{b}{k} \rfloor$. Било који број из интервала $[n_l, n_d]$ задовољава обе неједнакости и представља количник неког броја из интервала $[a, b]$ бројем k. Слично, било који број из интервала $[a, b]$ дељив бројем k даје неки количник из интервала $[n_l, n_d]$. Зато је тражени број бројева из интервала $[a, b]$ који су дељиви бројем k једнак броју бројева у интервалу $[n_l, n_d]$ а то је $n_d - n_l + 1$ ако је $n_d \geq n_l$, тј. 0 ако је тај интервал празан тј. ако је $n_d < n_l$. Бројеви n_l и n_d се могу одредити, на пример, гранањем.

Сложеност овог решења је, јасно, константна тј. $O(1)$.

```
#include <iostream>

using namespace std;

int main() {
  int a, b, k;
  cin >> a >> b >> k;
  int l = a % k == 0 ? a/k : a/k + 1; // ceil(a/k)
  int d = b / k;                      // floor(b/k)
  int broj = d >= l ? d-l+1 : 0;
  cout << broj << endl;
  return 0;
}
```

Још један начин да се задатак ефикасно реши је да се по узору на решење задатка Дељиви око броја одреди најмањи број већи или једнак броју a који је дељив бројем k и да се одреди највећи број мањи или једнак броју b који је дељив бројем k. Њиховим дељењем са k добијају се бројеви n_l и n_d описани у претходном решењу и задатак се даље решава на исти начин.

Сложеност овог решења је, јасно, константна тј. $O(1)$.

```
#include <iostream>
```

```cpp
using namespace std;

int main() {
  int a, b, k;
  cin >> a >> b >> k;
  int l = a % k == 0 ? a : (a/k + 1)*k;
  int d = b % k == 0 ? b : (b/k)*k; // moze i samo d = (b/k)*k;
  int broj = d >= l ? (d/k - l/k + 1) : 0;
  cout << broj << endl;
  return 0;
}
```

Задатак: Хлебови

Сваки човек носи два хлеба, жена један, а дете пола хлеба. Укупно их је n, и носе n хлебова. Написати програм који учитава природан број n и исписује колико има решења за број људи, жена и деце.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^9$).

Излаз: На стандардни излаз исписати број решења

Пример 1

Улаз	*Излаз*
5	2

Објашњење

Могућа решења су $(1, 2, 2)$ и $(0, 5, 0)$.

Пример 2

Улаз

```
12
```

Излаз

```
5
```

Задатак: Дељиви око броја

Дати су цели бројеви n и k. Напиши програм који одређује највећи цео број $l \leq n$ дељив бројем k и најмањи цео број $d \geq n$ дељив бројем k.

Улаз: Са стандардног улаза уносе се два цела броја (сваки у посебном реду).

- n ($0 \leq n \leq 100000$)
- k ($1 \leq k \leq 10000$)

Излаз: На стандардни излаз исписати два тражена цела броја, сваки у посебном реду.

Пример 1

Улаз	*Излаз*
23	20
5	25

Пример 2

Улаз	*Излаз*
49	49
7	49

Задатак: Максимални принос

Фармер поседује њиву димензије $a \times b$ метара. Да би лакше парцелисао њиву, бројеви a и b су цели. На основу субвенције добио је могућности да продужи странице своје њиве укупно за c метара (али тако да њива остане целобројних димензија). Он жели да то уради тако да након продужења површина буде што већа, тако да може да оствари што већи укупан принос. Напиши програм који одређује највећу могућу површину њиве након продужења страница.

Улаз: Са стандардног улаза се учитавају природни бројеви $a, b, c \leq 10^9$, раздвојени са по једним размаком.

Излаз: На стандардни излаз исписати максималну површину након продужења страница.

Пример 1

Улаз *Излаз*
`5 10 3` `80`
Објашњење

Димензија након проширења ће бити 8×10.

Пример 2

Улаз

`9 10 4`

Излаз

`132`

Објашњење

Димензија након проширења ће бити 11×12.

Пример 3

Улаз

`14 17 5`

Излаз

`324`

Објашњење

Димензија након проширења ће бити 18×18.

Решење

Груба сила

Задатак може бити решен грубом силом, тј. испробавањем свих могућности расподеле додатне дужине. За сваку дужину i између 0 и c, додајемо i страници a и $c - i$ страници b, израчунавамо површину и тражимо максимум тако добијених површина.

Сложеност овог решења је $O(c)$.

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
  long long a, b, c;
  cin >> a >> b >> c;
  long long maks = a * (b + c);
  for (long long i = 1; i <= c; i++)
    maks = max(maks, (a+i)*(b+c-i));
  cout << maks << endl;
}
```

Израчунавање максималног приноса

Од свих правоугаоника датог фиксираног обима, највећу површину има квадрат. Заиста, ако је познат обим правоугаоника $O = 2(a+b)$, тада је познат и полуобим $a+b = s$. Површина $a \cdot b = a \cdot (s-a) = a \cdot s - a \cdot a = s^2/4 - (a - s/2)^2$. Пошто је $(a - s/2)^2 \geq 0$, површина не може бити већа од $s^2/4$, а једнака је тој вредности када је $a = b = s/2$. Зато увећање треба направити тако да се добије облик који је што сличнији квадрату.

Нека је $a \leq b$. Ако је $a + c \leq b$, тада се целокупан износ увећања c може додати на мању страницу a. У супротном се прво краћа страница a продужи тако да постане једнака дужој страници b, а затим се преостали износ увећања $(c - (b - a))$ подели што равномерније могуће (ако је то паран број може се добити квадрат, а

ако није, тада се добија правоугаоник код којег је једна страница за један дужа од друге). У имплементацији дужине нових страница можемо израчунати тако што дужу страницу b увећамо за $\left\lfloor \frac{c-(b-a)}{2} \right\rfloor$ и за $\left\lceil \frac{c-(b-a)}{2} \right\rceil = \left\lfloor \frac{c-(b-a)+1}{2} \right\rfloor$.

Сложеност овог решења је $O(1)$.

```
#include <iostream>
#include <algorithm>

using namespace std;

int main() {
  long long a, b, c;
  cin >> a >> b >> c;
  if (a > b) swap(a, b);
  if (c <= b - a)
    a += c;
  else {
    long long preostalo = c - (b - a);
    a = b + preostalo / 2;
    b = b + (preostalo + 1) / 2;
  }
  cout << a * b << endl;
  return 0;
}
```

Задатак: Растављања на збир узастопних

Напиши програм који одређује на колико се начина дати природни број n може представити као збир два или више узастопна природна броја (већа или једнака 1).

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^9$).

Излаз: На стандардни излаз исписати тражени број начина.

Пример

Улаз	*Излаз*
15	3

Објашњење

$$15 = 1 + 2 + 3 + 4 + 5 = 4 + 5 + 6 = 7 + 8$$

Решење

Испробавање свих могућности за први члан, па за дужину

Први покушај може бити решавање проблема грубом силом, тј. испробавање свих могућих првих чланова збира. Најмањи могући први члан је $a_0 = 1$. Пошто збир мора да буде бар двочлан, највећи могући први члан је онај број a_0 такав да је $a_0 + (a_0 + 1) \leq n$. Када смо одредили први члан, одређујемо колико сабирака треба узети да би се добио збир n. Крећемо од двочланог низа и затим додајемо један по један наредни сабирак све док збир не достигне или не престигне збир n. Бројач увећавамо ако је након петље збир једнак вредности n (тада је успешно нађено једно решење).

Спољашња петља се извршава око $\frac{n}{2}$ пута. Број извршавања унутрашње петље је теже проценити. Питамо се који је број сабирака m потребан, тако да је $a_0 + (a_0 + 1) + \ldots + (a_0 + (m-1)) \geq n$. Ако применимо формулу за збир аритметичког низа, видимо да је тај збир једнак $m \cdot a_0 + \frac{m(m-1)}{2}$. Веома груба процена када је a_0 мало даје нам процену за m око $\sqrt{2n}$. Додуше, чим a_0 крене да расте, овај број крене да опада. Веома грубо, сложеност можемо ограничити одозго као $O(n\sqrt{n})$.

```cpp
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;
  int brojNacina = 0;
  for (int a0 = 1; a0 + (a0+1) <= n; a0++) {
    int zbir = a0 + (a0+1);
    for (int ai = a0 + 2; zbir < n; ai++)
      zbir += ai;
    if (zbir == n)
      brojNacina++;
  }
  cout << brojNacina << endl;
  return 0;
}
```

Испробавање свих могућности за дужину, па за први члан

Редослед петљи може бити другачији. Спољном петљом можемо одређивати број сабирака m, а унутрашњом испробавати вредности почетног сабирка. Крећемо од два сабирка. Највећи могући број сабирака наступа када је $a_0 = 1$, и пошто је $a_0 + (a_0 + 1) + \ldots + (a_0 + (m-1)) = m \cdot a_0 + \frac{m(m-1)}{2}$, да би збир могао да евентуално буде n мора да важи да је $m + \frac{m(m-1)}{2} \leq n$.

Сложеност је идентична као у претходном приступу и може се грубо проценити са $O(n\sqrt{n})$.

```cpp
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;
  int brojNacina = 0;
  for (int m = 2; m + m*(m-1)/2 <= n; m++) {
    int a0 = 1;
    int zbir = a0*m + m*(m-1)/2;
    while (zbir < n) {
      a0++;
      zbir = a0*m + m*(m-1)/2;
    }
    if (zbir == n)
      brojNacina++;
  }
  cout << brojNacina << endl;
  return 0;
}
```

Испробавање свих могућности за дужину и израчунавање првог члана

Кључна оптимизација наступа када увидимо да нам унутрашња петља није потребна. Наиме, нема потребе испробавати различите вредности a_0, већ се a_0 може израчунати на основу m и n. Ако је $m \cdot a_0 + \frac{m(m-1)}{2} = n$, тада је $a_0 = \frac{n - \frac{m(m-1)}{2}}{m}$. Збир са m сабирака постоји ако и само ако је ово цео број (што се може проверити испитивањем остатка при дељењу бројева $n - \frac{m(m-1)}{2}$ и m). Услов $m + \frac{m(m-1)}{2} \leq n$ гарантује да је $a_0 \geq 1$.

Напоменимо да је редослед провера био веома важан, јер је једначина линеарна по a_0, а квадратна по m, тако да је за фиксирано m, a прилично једноставно израчунати, док за фиксирано a није једноставно израчунати

m.

Сложеност једине петље, па и целог програма се може грубо осенити са $O(\sqrt{n})$ (у њеном телу се провера постојања броја a_0 врши у сложености $O(1)$).

```
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;
  int brojNacina = 0;
  for (int m = 2; m + m*(m-1)/2 <= n; m++)
    if ((n - m*(m-1)/2) % m == 0)
      brojNacina++;
  cout << brojNacina << endl;
  return 0;
}
```

Задатак: Пријатељски бројеви

Бројеви су пријатељски, ако је збир делилаца првог броја једнак другом, а збир делилаца другог броја једнак првом броју (у збир делилаца броја се убраја број 1, али не и сам тај број). Напиши програм који исписује све парове пријатељских бројева такве да оба броја леже у датом интервалу.

Улаз: Са стандардног улаза се учитавају бројеви a и b ($1 \leq a \leq b \leq 500000$).

Излаз: На стандардни излаз исписати све тражене парове уређене растући по првом елементу, тако је у сваком пару први број мањи или једнак од другога.

Пример 1

Улаз	*Излаз*
1	6 6
1000	28 28
	220 284
	496 496

Пример 2

Улаз	*Излаз*
300000	308620 389924
400000	356408 399592

Задатак: Највећи заједнички делилац

Мрави, пчеле и комарци организују спортски турнир и желе да се поделе у тимове, тако да се сваки тим састоји само од једне врсте инсеката, да сви тимови имају исти број чланова (да би се након рунде квалификација унутар сваке врсте могли своје представнике да пошаљу на заједнички турнир) и да је сваки инсект укључен тачно у један тим. Ако се зна број инсеката сваке од три дате врсте, напиши програм који одређује највећи могући број чланова сваког тима.

Улаз: Са стандардног улаза се уносе три броја из интервала $[1, 2 \cdot 10^9]$, сваки у посебном реду: број мрава, пчела и комараца.

Излаз: На стандардни излаз исписати један цео број - тражену величину тима.

Пример

Улаз	*Излаз*
20	10
30	
40	

Решење

Ако са a, b и c обележимо број сваке од три врсте инсеката, а са t величину сваког тима, бројеви a, b и c морају бити дељиви са t (јер сваки инсект мора бити укључен тачно у један тим). Дакле, тражимо највећи број t који дели бројеве a, b и c, а то је њихов највећи заједнички делилац (НЗД).

НЗД три броја се може одредити као НЗД од НЗД-а прва два и трећег броја, тј. важи $nzd(a, b, c) = nzd(nzd(a, b), c)$. Довољно је, дакле, да опишемо поступак одређивања НЗД два броја.

Еуклидов алгоритам

За одређивање НЗД најбоље је употребити чувени Еуклидов алгоритам. Оригинална формулација Еуклидовог алгоритма заснована је на одузимању.

Ако су два броја једнака тј. ако је $a = b$, тада им је и њихов НЗД једнак.

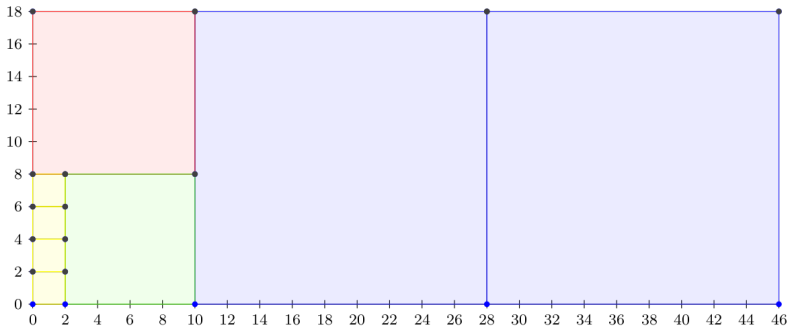
У супротном се проблем може свести на проналажење НЗД мањег од два броја и разлике два броја. На пример, ако је $a > b$, тада је $nzd(a, b) = nzd(a - b, b)$.

Заиста, ако неки број d дели бројеве a и b, тада он дели и њихову разлику. Дакле, $d = nzd(a, b)$ сигурно дели и b и $a - b$. Ако он не би био НЗД бројева $a - b$ и b, тада би постојао неки већи број d' који би делио и $a - b$ и b. Међутим, тада би d' делио и збир $a = (a - b) + b$, па би био делилац бројева a и b, који је већи од d, што је контрадикција са тим да је d НЗД бројева a и b.

Ако је $a < b$, тада се веома слично може доказати да је $nzd(a, b) = nzd(a, b - a)$.

Алгоритам се може илустровати и геометријски (и у директној је вези са проблемом одређивања максималне димензије квадрата којима може да се поплоча правоугаоно поље димензија $a \times b$.

Претпоставимо да је дат правоугаоник чије су дужине страница a и b и да је потребно одредити највећу дужину странице квадрата таква да се правоугаоник може поплочати квадратима те димензије. Ако је полазни правоугаоник димензије $a = 46$ и $b = 18$, тада се прво из њега могу исећи два квадрата димензије 18 пута 18 и остаће нам правоугаоник димензије 18 пута 10. Јасно је да ако неким мањим квадратима успемо да поплочамо тај преостали правоугаоник, да ћемо тим квадратима успети да поплочамо и ове квадрате димензије 18 пута 18 (јер ће димензија тих малих квадрата делити број 18), па ћемо самим тим моћи поплочати и цео полазни правоугаоник димензија 46 пута 18. Од правоугаоника димензије 18 пута 10 можемо исећи квадрат димензије 10 пута 10 и преостаће нам правоугаоник димензије 10 пута 8. Поново, квадратићи којима ће се моћи поплочати тај преостали правоугаоник ће бити такви да се њима може поплочати и исечени квадрат димензије 10 пута 10. Од тог правоугаоника исецамо квадрат 8 пута 8 и добијамо правоугаоник димензије 8 пута 2. Њега можемо разложити на четири квадрата димензије 2 пута 2 и то је највећа димензија квадрата којим се може поплочати полазни правоугаоник.



Имплементација може бити рекурзивна, која директно прати претходну дефиницију. Могуће је једноставно направити и итеративну имплементацију, у којој се у сваком кораку вредност већег од два броја мења њиховом разликом.

Најгори случај наступа када је разлика између два броја јако велика (ако је $a = 1$, он ће се одузимати од b све док он не постане 1, за шта је потребно $b - 1$ корака). Може се показати да је сложеност линеарна у односу на већи од два броја, што се може записати као $O(\max(a, b))$ или $O(a + b)$. Ако се сложеност рачуна у односу на број цифара броја (што је величина улаза), онда је она заправо експоненцијална.

```
#include <iostream>

using namespace std;

// izracunavanje najveceg zajednickog delioca brojeva a i b
int nzd(int a, int b) {
  // Euklidov algoritam sa oduzimanjem
  while (a != b) {
    if (a > b)
      a -= b;
```

```
    else
      b -= a;
  }
  return a;
}

int main() {
  // ucitavamo 3 broja
  int a, b, c;
  cin >> a >> b >> c;
  // izracunavamo njihov nzd
  cout << nzd(nzd(a, b), c) << endl;
  return 0;
}
```

Поправљање сложености коришћењем дељења

Размотримо одређивање НЗД на једном примеру.

$nzd(279, 45) = nzd(234, 45) = nzd(189, 45) = nzd(144, 45) = nzd(99, 45) = nzd(54, 45) = nzd(9, 45) = nzd(9, 36) = nzd(9, 27) = nzd(9, 18) = nzd(9, 9) = 9$.

Можемо приметити дугачак низ корака у којима се мало по мало од броја 279 одузима број 45, све док се не дође до броја који је мањи од броја 45. За тим нема потребе, јер знамо да ће после тог дугог низа поступак зауставити када нам један аргумент буде баш 45, а други буде једнак остатку при дељењу броја 279 бројем 45, а то је број 9. Дакле, уместо да итеративно, тај остатак рачунамо узастопним одузимања, бољи приступ је да применимо дељење и у једном кораку га израчунамо као остатак при дељењу. Ако сличан принцип применимо на бројеве 9 и 45, доћи ћемо до тога да ће нам остати број 9 и остатак при дељењу та два броја, што је нула. То није баш у потпуности једнако као у случају одузимања, где смо се зауставили код пара $(9, 9)$, међутим, сасвим је исправно и може се сматрати продужењем претходног поступка у ком би се пре пријављивања резултата урадило још једно одузимање и дошло се до тога да је један од бројева једнак нули, када је НЗД једнак другом броју.

Бржи Еуклидов алгоритам, у ком се користи дељење, заснован је на следећим тврђењима.

- НЗД било ког броја a и нуле је тај број a (он дели и нулу и самог себе и највећи је такав број јер није могуће да неки број већи од a дели број a).
- НЗД бројева a и b, када b није нула, једнак је НЗД броја b и остатка при дељењу a бројем b, тј. $nzd(a, b) = nzd(b, a \bmod b)$.

Приметимо да нема потребе анализирати који је број мањи, а који је већи. Ако је $a < b$, тада ће важити $a \bmod b = a$, па ће се у првом кораку добити да је $nzd(a, b) = nzd(b, a)$. Пошто је $a \bmod b < b$, једном када је први аргумент већи од другог, то ће тако остати до краја.

Докажимо формално коректност алгоритма. На основу дефиниције целобројног дељења важи да је $a = (a \operatorname{div} b) \cdot b + (a \bmod b)$. Обележимо са d НЗД бројева b и $a \bmod b$. Да бисмо доказали да је он уједно НЗД бројева a и b довољно је доказати да он дели та два броја и да сваки број који дели та два броја дели њега.

- Пошто број d дели бројеве b и $a \bmod b$, он дели оба сабирка на десној страни, па зато дели и њихов збир који је једнак a и зато дели и a и b.
- Даље, ако неки број d' дели бројеве a и b он мора делити и број $a \bmod b$ (јер се он може исказати као разлика два броја дељивих бројем d'), па пошто је d' делилац бројева b и $a \bmod b$, он мора делити и њихов НЗД тј. мора делити број d.

Еуклидов алгоритам, дакле, допушта веома једноставну рекурзивну карактеризацију.

Имплементацију Еуклидовог алгоритма можемо извршити итеративно, тако што у петљи која се извршава све док је број b већи од нуле пар променљивих (a, b) мењамо вредностима $(b, a \bmod b)$. Наивни покушај да се то уради на следећи начин:

```
a = b;
b = a % b;
```

није коректан, јер се приликом израчунавања остатка користи промењена вредност промењиве `a`. Зато је неопходно употребити помоћну променљиву. На пример:

```
tmp = b;
b = a % b;
a = tmp;
```

На крају петље вредност b једнака је нули, тако да као резултат можемо пријавити текућу вредност броја a.

Еуклидов алгоритам који је заснован на дељењу можемо представити и на следећи начин:

$$\begin{aligned}
r_0 &= a \\
r_1 &= b \\
r_2 &= r_0 - q_1 r_1, & 0 < r_2 < r_1 \\
&\dots \\
r_{i+1} &= r_{i-1} - q_i r_i, & 0 < r_{i+1} < r_i \\
&\dots \\
r_k &= r_{k-2} - q_{k-1} r_{k-1}, & 0 < r_k < r_{k-1} \\
r_{k+1} &= r_{k-1} - q_k r_k, & 0 = r_{k+1}
\end{aligned}$$

Вредност r_k је НЗД бројева a и b. Заиста, пошто је $r_{k+1} = 0$, важи да је $r_{k-1} = q_k r_k$, па број r_k дели r_{k-1}. Међутим, пошто је $r_{k-2} = q_{k-1} r_{k-1} + r_k$, он дели и r_{k-2}. Сличним резоновањем, уназад, може се закључити да r_k дели и r_1 и r_0 тј. a и b. Обратно, ако неки број дели и a и b онда он дели и r_0 и r_1, а пошто је $r_2 = r_0 - q_1 r_1$, он дели и r_2. Сличним резоновањем, унапред, може се закључити да тај број мора делити и r_k. Стога је r_k НЗД бројева a и b.

У сваком кораку алгоритма одржавамо две узастопне вредности низа r_i. У почетку, то су чланови r_0 и r_1 тј. оригиналне вредности a и b. Важи да је $q_1 = a \operatorname{div} b$, а $r_2 = a \bmod b$. У другом кораку промењиве a и b треба да имају вредности чланова r_1 и r_2, што значи да се пар промењивих a, b мења вредностима b и $a \bmod b$, што је управо оно што смо радили у претходно описаном коду. Поступак се наставља све док пар узастопних вредности не постане r_k, r_{k+1}, тј., пошто пар узастопних вредности одржавамо у промењивама a и b, док b не постане нула и тада је НЗД који је једнак r_k садржан у промењивој a.

Приметимо да се после највише једног корака осигурава да је $a > b$ (јер се у сваком кораку пар (a, b) мења паром $(b, a \bmod b)$, а увек важи да је $a \bmod b < b$). После било које две итерације се од пара (a, b) долази до пара $(a \bmod b, b \bmod (a \bmod b))$ (наравно, под претпоставком да је $b \neq 0$ и да је $a \bmod b \neq 0$). Докажимо да је $a \bmod b < a/2$. Ако је $b \leq a/2$, тада је $a \bmod b < b \leq a/2$. У супротном, за $b > a/2$ важи да је $a \bmod b = a - b < a/2$. Зато се први аргумент после свака два корака смањи бар двоструко. До вредности 1 први аргумент стигне у логаритамском броју корака у односу на већи од полазна два броја и тада други број сигурно достиже нулу (јер је строго мањи од првог) и поступак се завршава. Дакле, сложеност је логаритамска у односу на већи од два броја, што може да се запише и као $O(\log(a + b))$. Ако се сложеност рачуна у односу на број цифара броја (што је величина улаза), онда је она заправо линеарна.

```
#include <iostream>

using namespace std;

// izracunavanje najveceg zajednickog delioca brojeva a i b
int nzd(int a, int b) {
  // Euklidov algoritam
  while (b > 0) { // dok b ne postane nula
    int tmp = b;  // par (a, b) menjamo parom (b, a % b)
    b = a % b;    // jer je nzd(a, b) = nzd(b, a % b)
    a = tmp;
  }
  return a;       // nzd(a, 0) = a
}

int main() {
  // ucitavamo 3 broja
  int a, b, c;
```

```
  cin >> a >> b >> c;
  // izracunavamo njihov nzd
  cout << nzd(nzd(a, b), c) << endl;
  return 0;
}
```

2.7 Одсецање

Један од основних принципа за добијање ефикаснијих алгоритама и програма је да рачунар не треба да израчунава ствари за које се унапред може проценити да нису потребне за добијање коначног решења проблема. Важан пример овог принципа се јавља код алгоритама претраге. Претрагу елемената не треба експлицитно вршити међу елементима за које се може унапред утврдити да не могу да задовоље услов претраге. Када прескочимо проверу таквих елемената, кажемо да смо учинили **одсецање у претрази**. Сличан принцип се примењује и када се врши оптимизација (тражи најмањи односно највећи елемент), када се може прескочити анализа елемената за које се унапред може доказати да су мањи (односно већи) од траженог максимума (односно минимума).

Да би се осигурала коректност алгоритама у којима се врши одсецање увек је потребно веома пажљиво утврдити да је одсецање оправдано и да се у делу простора претраге који се не испитује заиста не може налазити решење проблема.

У наставку овог поглавља ћемо кроз одређен број примера приказати како се одсецањем постиже асимптотски ефикаснији алгоритам. Један од најзначајнијих примера одсецања представља *бинарна претрага*, која ће, због свог значаја бити анализирана у посебном поглављу. Одсецање се примењује и у другим облицима претраге (бектрекинг претрази, претрази у дубину, претрази у ширину), о чему ће више бити речи у каснијим поглављима.

Задатак: Прост број

Напиши програм који испитује да ли је унети природан број прост (већи је од 1 и нема других делилаца осим 1 и самог себе).

Улаз: Са стандардног улаза се уноси природан број n ($1 \leq n \leq 10^9$).

Излаз: На стандардни излаз исписати `DA` ако је број n прост тј. `NE` ако није.

Пример		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`17`	`DA`	`903543481`	`NE`

Решење

Линеарна претрага свих потенцијалних делилаца

Природан број је прост ако је већи од 1 и ако није дељив ни са једним бројем осим са један и са самим собом. По дефиницији број 1 није прост. Дакле, број већи од 1 је прост ако нема ни једног правог делиоца. Потребно је дакле извршити претрагу скупа потенцијалних делилаца и проверити да ли неки од њих стварно дели број n. Имплементација се заснива на алгоритму линеарне претраге. Тај смо алгоритам описали, на пример, у задатку Негативан број. Скуп потенцијалних делилаца је скуп свих природних бројева од 2 до $n-1$ и наивна имплементација их све проверава.

Пошто се провера сваког делиоца извршава израчунавањем једног остатка при дељењу, сложеност овог приступа одговара броју делилаца и једнака је $O(n)$.

```
#include <iostream>

using namespace std;

bool prost(int n) {
  if (n == 1) return false;
  for (int i = 2; i < n; i++)
    if (n % i == 0)
      return false;
```

```
  return true;
}

int main() {
  int n;
  cin >> n;
  cout << (prost(n) ? "DA" : "NE") << endl;
  return 0;
}
```

Одсецање у претрази

Делиоци броја се увек јављају у пару. На пример, делиоци броја 100 организовани по паровима су (1, 100), (2, 50), (4, 25) (5, 20) и (10, 10). Ако је i делилац броја n, делилац је и број $\frac{n}{i}$. При том, ако је $i \geq \sqrt{n}$, тада је $\frac{n}{i} \leq \sqrt{n}$. Дакле, важи следећа теорема.

Теорема. Природан број $n \geq 2$ има праве делиоце који су већи или једнаки вредности $\sqrt{n}$ ако и само ако има делиоце који су мањи или једнаки вредности $\sqrt{n}$.

Ова теорема нам даје могућност да претрагу потенцијалних делилаца редукујемо само на интервал $[2, \sqrt{n}]$, јер ако број нема делилаца мањих или једнаких вредности $\sqrt{n}$, онда не може да има делилаца већих или једнаких тој вредности, тј. нема правих делилаца и прост је. Ово је пример алгоритма у ком се ефикасност значајно поправља тако што је елиминисан (одсечен) значајан део простора претраге за који можемо да докажемо да га није неопходно проверавати.

Сама имплементација је једноставна и заснива се на алгоритму линеарне претраге. У посебној функцији на почетку проверавамо специјалан случај броја 1 (ако је n једнако 1, враћамо вредност `false`). Након тога, у петљи проверавамо потенцијалне делиоце од 2 до $\sqrt{n}$. Један начин да одредимо горњу границу је да употребимо библиотечку функцију `sqrt`. Међутим, рад са реалним бројевима је могуће у потпуности избећи тако што се уместо услова $i \leq \sqrt{n}$ употреби услов $i \cdot i \leq n$. За сваку вредност i проверава се да ли је делилац броја i (израчунавањем остатка при дељењу). Чим се утврди да је i делилац броја n функција може да врати `false` (тиме се уједно прекида извршавање петље). На крају петље, функција може да врати `true`, јер није пронађен ниједан делиоц мањи или једнак од $\sqrt{n}$, па на основу теореме које смо доказали не може постојати ни један делилац изнад те вредности и број је прост.

Сложеност овог алгоритма је $O(\sqrt{n})$. Обратите пажњу на то да је ово скраћивање интервала претраге веома значајно (ако је највећи број око 10^9 тј. око милијарду, уместо милијарду делилаца потребно је проверавати само њих корен из милијарду, што је тек нешто изнад тридесет хиљада).

```
#include <iostream>

using namespace std;

bool prost(int n) {
  if (n == 1) return false;
  for (int i = 2; i*i <= n; i++)
    if (n % i == 0)
      return false;
  return true;
}

int main() {
  int n;
  cin >> n;
  cout << (prost(n) ? "DA" : "NE") << endl;
  return 0;
}
```

Могуће је правити и другачије имплементације истог алгоритма.

```
#include <iostream>
```

```
using namespace std;

bool prost(int n) {
  int i = 2;
  while (i*i <= n && n % i != 0)
    i++;
  return n > 1 && i*i > n;
}

int main() {
  int n;
  cin >> n;
  cout << (prost(n) ? "DA" : "NE") << endl;
  return 0;
}
```

Провера само непарних бројева

Још једна могућа оптимизација је да се на почетку провери да ли је број паран а да се након тога проверавају само непарни делиоци, међутим, та оптимизација не доноси превише (обилазак до корена смањује број потенцијалих кандидата са милијарде на тек тридесетак хиљада, а провера само непарних делилаца тај број смањује на петнаестак хиљада, што је сразмерно знатно мања уштеда).

Сложеност овог алгоритма је $O(\sqrt{n})$, али се провером само непарних бројева константни фактор смањио два пута.

```
#include <iostream>

using namespace std;

// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;     // broj 1 nije prost
  if (n == 2) return true;      // broj 2 jeste prost
  if (n % 2 == 0) return false; // ostali parni brojevi nisu prosti
  // proveravamo neparne delioce od 3 do korena iz n
  for (int i = 3; i*i <= n; i += 2)
    if (n % i == 0)
      return false;
  // nismo nasli delioca - broj jeste prost
  return true;
}

int main() {
  int n;
  cin >> n;
  cout << (prost(n) ? "DA" : "NE") << endl;
  return 0;
}
```

Провера само бројева облика 6k-1 и 6k+1

Програм се још мало може убрзати ако се примети да су сви прости бројеви већи од 2 и 3 облика $6k-1$ или $6k+1$, за $k \geq 1$ (наравно, обратно не важи). Заиста, бројеви облика $6k$, $6k+2$ и $6k+4$ су сигурно парни тј. дељиви са 2, бројеви облика $6k+3$ су дељиви са 3, тако да су једини преостали $6k+1$ и $6k+5$, при чему су ови други сигурно облика $6k'-1$ (за $k'=k+1$). Дакле, уместо да проверавамо дељивост са свим непарним бројевима мањим од корена, можемо проверавати дељивост са свим бројевима облика $6k-1$ или $6k+1$, чиме избегавамо проверу са једним на свака три непарна броја и програм убрзамо сходно томе.

Сложеност овог алгоритма је $O(\sqrt{n})$, али се провером само бројева облика 6k-1 и 6k+1 константни фактор смањио три пута у односу на први алгоритам ове сложености.

```
#include <iostream>
using namespace std;

bool prost(int n) {
  if (n == 1 ||
      (n % 2 == 0 && n != 2) ||
      (n % 3 == 0 && n != 3))
    return false;
  for (int k = 1; (6*k - 1) * (6*k - 1) <= n; k++)
    if (n % (6 * k + 1) == 0 || n % (6 * k - 1) == 0)
      return false;
  return true;
}

int main() {
  int n;
  cin >> n;
  cout << (prost(n) ? "DA" : "NE") << endl;
}
```

Ако је потребно за више бројева одједном проверити да ли су прости, уместо проверавања сваког појединачног, боље је употребити Ератостеново сито. Тај алгоритам је описан у задатаку Ератостеново сито.

Задатак: Ератостеново сито

Напиши програм који одређује број простих бројева у интервалу $[a, b]$ и њихов збир (ако збир има више од 6 цифара, исписати само последњих 6).

Улаз: Са стандардног улаза уносе се бројеви a и b ($1 \leq a \leq b \leq 10^7$), сваки у посебној линији.

Излаз: На стандардном излазу приказати у једној линији, одвојени једним бланко знаком, број простих бројева из интервала $[a, b]$ и тражени збир.

Пример

Улаз	*Излаз*
1 1000	168 76127

Решење

Појединачне провере простих бројева

Очигледан алгоритам за одређивање свих простих бројева из неког интервала јесте да се за сваки број из тог интервала појединачно провери да ли је прост. Ово може бити урађено уз помоћ алгоритма тј. функције коју смо описали у задатку Прост број.

На основу спецификације задатка потребно је одредити највише 6 последњих цифара збира свих простих бројева из интервала $[a, b]$, што, је еквивалентно одређивању збира тих бројева по модулу 10^6. Наиме, важи $(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$. У петљи пролазимо кроз све бројеве од a до b, вршимо филтрирање на основу услова да је број прост и вршимо бројање и сабирање добијене филтриране серије.

Напоменимо да се збир рачуна тако што се на почетку иницијализује на нулу, а затим се у сваком кораку израчунава сабирање збира и текућег простог броја по модулу 10^6 (`zbir = (zbir % 1000000 + p % 100000) % 100000`). Пошто ће у сваком кораку збир бити мањи од 10^6, и пошто не постоји опасност од прекорачења када се у обзир узме максимална вредност простих бројева који се сабирају, претходни корак се може заменити кораком `zbir = (zbir + p) % 1000000`.

Ако се провера да ли је дати број k прост врши у сложености $O(\sqrt{k})$, тада је овај алгоритам сложености $O((b - a)\sqrt{b})$. Ако је интервал облика $[0, n]$, сложеност је $O(n\sqrt{n})$.

```
#include <iostream>
#include <vector>

using namespace std;
```

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;     // broj 1 nije prost
  if (n == 2) return true;      // broj 2 jeste prost
  if (n % 2 == 0) return false; // ostali parni brojevi nisu prosti
  // proveravamo neparne delioce od 3 do korena iz n
  for (int i = 3; i*i <= n; i += 2)
    if (n % i == 0)
      return false;
  // nismo nasli delioca - broj jeste prost
  return true;
}

int main() {
  // ucitavamo granice intervala
  int a, b;
  cin >> a >> b;

  // odredjujemo broj i zbir po modulu 1000000 prostih brojeva iz
  // intervala [a, b]
  int zbir = 0, broj = 0;
  for (int i = a; i <= b; i++)
    if (prost(i)) {
      zbir = (zbir + i) % 1000000;
      broj++;
    }

  // prijavljujemo rezultat
  cout << broj << " " << zbir << endl;
  return 0;
}
```

Ератостеново сито

Бољи резултат од испитивања за сваки број појединачно да ли је прост може се добити применом алгоритма познатог као Ератостеново сито. Основна идеја алгоритма је да се прво напишу сви бројеви од 1 до датог броја n, затим да се прецрта број 1 (јер он по дефиницији није прост), након њега сви умношци броја 2 (нису прости зато што су дељиви са 2, док број 2 остаје непрецртан јер је он прост), затим умношци броја 3 (нису прости јер су дељиви бројем 3), затим умношци броја 5 (нису прости зато што су дељиви бројем 5) и тако даље.

Ефикасна имплементација овог алгоритма подразумева неколико одсецања (којима се избегава понављање истих операција више пута и асимптотски убрзава алгоритам).

Прво, умношке сложених бројева нема потребе посебно прецртавати јер су они већ прецртани током прецртавања умножака неког од њихових простих фактора (на пример, нема потребе посебно прецртавати умношке броја 4 јер су они већ прецртани током прецртавања умножака броја 2).

Друго, приликом прецртавања умножака броја d довољно је кренути од $d \cdot d$ јер су мањи умношци већ прецртани раније (сви имају праве факторе мање од d). Зато је потребно је да се поступак понавља само док се не прецртају умношци свих простих бројева који нису већи од корена броја n. За бројеве веће од корена од n прецртавање би кренуло од њиховог квадрата који је већи од n, па је јасно да се ни за један од њих ништа додатно не би прецртало.

Бројеви који су остали непрецртани су прости (јер знамо да немају правих делилаца мањих или једнаких корену од n, па самим тим и мањих или једнаких свом корену, а пошто немају делилаца испод вредности корена, немају правих делилаца ни изнад вредности корена). Та теорема је доказана у задатку Прост број.

Прикажимо како се овим алгоритмом одређују сви прости бројеви од 2 до 50. Крећемо од пуне табеле у којој су уписани сви бројеви од 2 до 50.

```
 .  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

У првом кораку прецртавамо све умношке броја 2 (осим самог броја 2).

```
 .  2  3  .  5  .  7  .  9  .
11  . 13  . 15  . 17  . 19  .
21  . 23  . 25  . 27  . 29  .
31  . 33  . 35  . 37  . 39  .
41  . 43  . 45  . 47  . 49  .
```

У наредном кораку прецртавамо све умношке броја 3, кренувши од његовог квадрата тј. од 9 (број 6 је већ прецртан као умножак броја 2).

```
 .  2  3  .  5  .  7  .  .  .
11  . 13  .  .  . 17  . 19  .
 .  . 23  . 25  .  .  . 29  .
31  .  .  . 35  . 37  .  .  .
41  . 43  .  .  . 47  . 49  .
```

Умношке броја 4 не прецртавамо, јер је он прецртан (па самим тим и сви његови умношци).

У наредном кораку прецртавамо све умношке броја 5, кренувши од његовог квадрата тј. броја 25 (умношци $2 \cdot 5$, $3 \cdot 5$ и $4 \cdot 5$ су већ прецртани).

```
 .  2  3  .  5  .  7  .  .  .
11  . 13  .  .  . 17  . 19  .
 .  . 23  .  .  .  .  . 29  .
31  .  .  .  .  . 37  .  .  .
41  . 43  .  .  . 47  . 49  .
```

Умношке броја 6 не прецртавамо, јер је он прецртан (па самим тим и сви његови умношци).

Прецртавамо умношке броја 7, кренувши од његовог квадрата тј. броја 49.

```
 .  2  3  .  5  .  7  .  .  .
11  . 13  .  .  . 17  . 19  .
 .  . 23  .  .  .  .  . 29  .
31  .  .  .  .  . 37  .  .  .
41  . 43  .  .  . 47  . 49  .
```

Умношке броја 8 не прецртавамо, јер је он прецртан (па самим тим и сви његови умношци). Међутим, прецртавање би кренуло од његовог квадрата. И за све наредне бројеве прецртавање креће од њиховог квадрата, међутим, тај квадрат је већ ван табеле (јер је већи од 50), па се поступак може завршити. Преостали бројеви су прости.

Прецртавање бројева моделоваћемо низом (или вектором) који садржи логичке вредности (вредности типа `bool`) и прецртане бројеве означаваћемо са `false`, а непрецртане са `true`. Одређивање простих бројева (помоћу поменутог низа тј. вектора) реализоваћемо у засебној функцији, јер та функција може бити корисна и у многим наредним задацима.

Рецимо и да је без обзира на то што су нама потребни само бројеви из интервала од a до b, у Ератостеновом ситу потребно вршити анализу свих бројева из интервала од 0 до b (јер се прецртавање мора вршити и бројевима мањим од a).

Анализа сложености је компликованија и захтева одређено (додуше веома елементарно) познавање теорије бројева. Проценимо број извршавања тела унутрашње петље. У почетном кораку спољне петље прецртава се око $\frac{n}{2}$ елемената. У наредном, око $\frac{n}{3}$. У наредном кораку је број 4 већ прецртан, па се не прецртава ништа. У наредном се прецртава око $\frac{n}{5}$, након тога опет ништа, затим $\frac{n}{7}$ итд. У последњем кораку се прецртава око $\frac{n}{\sqrt{n}}$ елемената. Дакле, број прецртавања је највише

$$\frac{n}{2}+\frac{n}{3}+\frac{n}{5}+\ldots+\frac{n}{\sqrt{n}}=n\cdot\left(\sum_{\substack{d\text{ prost},\\ d\leq\sqrt{n}}}\frac{1}{d}\right)$$

Број је заправо и мањи, јер приликом прецртавања у унутрашњој петљи прецртавање не крећемо од d, већ од d^2, али за потребе лакшег одређивања горње границе сложености користићемо претходну оцену.

Још је велики Ојлер открио да се збир $H(m)=1+1/2+1/3+\ldots+1/m=\sum_{d\leq m}\frac{1}{d}$ (такозвани хармонијски збир) асимптотски понаша слично функцији $\log m$ (разлика између ове две функције тежи такозваној Ојлер-Маскеронијевој константи $\gamma\approx 0.5772156649$), па самим тим знамо да тај збир дивергира. Такође, открио је да када се сабирање врши само по простим бројевима, тада се збир понаша као логаритам хармонијског збира, тј. као $\log\log m$ (па је и он дивергентан). Дакле, у нашем примеру можемо закључити да је број прецртавања једнак $n\cdot\log\log\sqrt{n}$. Пошто је $\log\log\sqrt{n}=\log\log n^{\frac{1}{2}}=\log\left(\frac{1}{2}\log n\right)=\log\frac{1}{2}+\log\log n$, под претпоставком да је сабирање бројева (које се користи у имплементацији петљи) константне сложености, важи да је сложеност Ератостеновог сита $O(n\cdot\log\log n)$. Иако није линеарна, функција $\log\log n$ толико споро расте, да се за све практичне потребе Ератостаново сито може сматрати линеарним у односу на n (што је доста спорије само од испитивања да ли је број n прост, што има сложеност $O(\sqrt{n})$, али је брже од проверавања сваког броја појединачно које је сложености $O(n\sqrt{n})$).

```
#include <iostream>
#include <vector>

using namespace std;

// funkcija koja popunjava logicki niz podacima o prostim brojevima iz
// intervala [0, n]
void Eratosten(vector<bool>& prost, int n) {
  // alociramo potreban prostor
  prost.resize(n + 1, true);
  prost[0] = prost[1] = false; // 0 i 1 po definiciji nisu prosti
  // brojevi ciji se umnosci precrtavaju
  for (int i = 2; i * i <= n; i++)
    // nema potrebe precrtavati umnoske slozenih brojeva
    if (prost[i]) {
      // precrtavamo umnoske broja i i to krenuvsi od i*i
      for (int j = i * i; j <= n; j += i)
        prost[j] = false;
    }
}

int main() {
  // ucitavamo granice intervala
  int a, b;
  cin >> a >> b;

  // odredjujemo proste brojeve u intervalu [0, b]
  vector<bool> prost;
  Eratosten(prost, b);

  // odredjujemo broj i zbir po modulu 1000000 prostih brojeva iz
  // intervala [a, b]
  int zbir = 0, broj = 0;
  for (int i = a; i <= b; i++)
    if (prost[i]) {
      zbir = (zbir + i) % 1000000;
      broj++;
    }
```

```
  // prijavljujemo rezultat
  cout << broj << " " << zbir << endl;
  return 0;
}
```

Задатак: Збир простих прост

Напиши програм који за дати природан број n одређује колико има парова простих бројева (p, q) таквих да је $p < q$ и $p + q \leq n$ је такође прост.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^6$).

Излаз: На стандардни излаз исписати тражени број парова, такав да је $p + q \leq n$.

Пример 1

Улаз *Излаз*
6 1

Објашњење

Једини пар који задовољава услове је $(2, 3)$, јер је $5 \leq 6$ прост број.

Пример 2

Улаз

100

Излаз

8

Решење

Провера свих парова

Решење грубом силом подразумева да се за сваки пар бројева $2 \leq p < q \leq n - p$ провери да ли испуњава услове задатка. Набрајање парова можемо вршити угнежђеним петљама, а проверу да ли је број прост алгоритмом описаним у задатку Прост број.

Пошто се провера да ли је дати број прост реализује у времену $O(\sqrt{n})$, а таквих парова има $O(n^2)$, сложеност овог приступа је $O(n^2\sqrt{n})$.

Једну малу оптимизацију можемо постићи коришћењем одсецања – ако број p није прост, нема потребе да покрећемо унутрашњу петљу и да проверавамо различите вредности q. Пошто простих бројева мањих од n има отприлике $\frac{n}{\log n}$, овим се програм убрзава асимптотски, али, нажалост, само за фактор $\log n$.

```
#include <iostream>
#include <vector>

using namespace std;

bool prost(int n) {
  if (n < 2)
    return false;
  for (int d = 2; d * d <= n; d++)
    if (n % d == 0)
      return false;
  return true;
}

int main() {
  int n;
  cin >> n;

  int brojParova = 0;
```

```
  for (int p = 2; p <= n; p++)
    if (prost(p))
      for (int q = p+1; p+q <= n; q++)
        if (prost(q) && prost(p+q))
          brojParova++;

  cout << brojParova << endl;

  return 0;
}
```

Ератостеново сито и провера свих парова

Одређивање свих простих бројева из интервала $[0, n]$ можемо урадити ефикасно Ератостеновим ситом (као у задатку Ератостеново сито). Након тога за све парове $2 \leq p < q \leq n - p$ можемо проверити да ли задовољавају услов тако што проверу простости јако брзо вршимо на основу низа који је изгенерисан поступком Ератостеновог сита.

Сложеност Ератостеновог сита је $O(n \log \log n)$. Након конструкције низа, за сваки елемент можемо у у константној сложености проверити да ли је прост. Међутим, парова кандидата има $O(n^2)$, па временом извршавања доминира фаза провере свих парова и укупна сложеност је $O(n^2)$.

```
#include <iostream>
#include <vector>

using namespace std;

vector<bool> eratosten(int n) {
  vector<bool> prost(n+1, true);
  prost[0] = prost[1] = false;
  for (int d = 2; d * d <= n; d++)
    if (prost[d])
      for (int j = d * d; j <= n; j += d)
        prost[j] = false;
  return prost;
}

int main() {
  int n;
  cin >> n;
  vector<bool> prost = eratosten(n);

  int brojParova = 0;
  for (int p = 2; p <= n; p++)
    if (prost[p])
      for (int q = p+1; p+q <= n; q++)
        if (prost[q] && prost[p+q])
          brojParova++;

  cout << brojParova << endl;

  return 0;
}
```

Одсецање на основу парности

Ефикасност се значајно може поправити одсецањем. Наиме, може се приметити да ако су p и q прости бројеви већи од 2, они су непарни, а збир два непарна броја је паран број, који је сигурно већи од 2. Зато њихов збир не може да буде прост. Одатле следи да број p сигурно мора да буде једнак 2, а питање се своди на

проналажење свих бројева $2 < q \leq n - 2$, таквих да су q и $q + 2$ прости (то су такозвани прости бројеви близанци.

Одређивање свих простих бројева мањих или једнаких од n можемо урадити Ератостеновим ситом, у времену $O(n \log \log n)$. Након тога проналазак парова близанаца можемо урадити у додатном времену $O(n)$, тако да Ератостеново сито доминира целим алгоритмом.

```
#include <iostream>
#include <vector>

using namespace std;

vector<bool> eratosten(int n) {
  vector<bool> prost(n+1, true);
  prost[0] = prost[1] = false;
  for (int d = 2; d * d <= n; d++)
    if (prost[d])
      for (int j = d * d; j <= n; j += d)
        prost[j] = false;
  return prost;
}

int main() {
  int n;
  cin >> n;
  vector<bool> prost = eratosten(n);

  int brojParova = 0;
  int p = 2;
  for (int q = p+1; p+q <= n; q++)
    if (prost[q] && prost[p+q])
      brojParova++;

  cout << brojParova << endl;

  return 0;
}
```

Задатак: Серија сјајних партија

Кошаркаш има сјајну партију ако на њој да бар p поена. Напиши програм који на основу броја поена датих на свакој утакмици током сезоне одређује да ли је кошаркаш некада имао серију од више од k сјајних узастопних партија.

Улаз: Са стандардног улаза се учитава број p ($5 \leq p \leq 60$), након тога потребна дужина серије k, а затим број утакмица n ($3 \leq k \leq n \leq 10^5$). У наредних n редова учитава се број постигнутих поена (природан број мањи или једнак од 100).

Излаз: На стандардни излаз исписати `da` ако серија постоји тј. `ne` ако не постоји.

Пример 1

Улаз	*Излаз*
30	da
3	
5	
38	
30	
32	
28	
35	

Пример 2

Улаз	*Излаз*
30	ne
3	
5	
38	
30	
28	
32	
35	

Решење

Овај задатак је веома сличан задатку Најдужа серија победа и може се решавати свим техникама приказаним у том задатку.

Груба сила

Задатак се може решавати грубом силом, директно по дефиницији. То подразумева да се испитају сви могући узастопни низови од k утакмица и да се провери да ли неки од њих садржи само сјајне партије играча.

Овај приступ је прилично наиван и сложеност је $O(n-k) \cdot k$, што је $O(n^2)$ ако је k приближно једнако половини низа. Наиме анализира се засебно $n-k$ низова дужине k.

```
#include <iostream>
#include <vector>

using namespace std;

// provera da li postoji serija potrebne duzine u kojoj je takmicar
// postigao potreban broj poena
bool postojiSerija(const vector<int>& postigaoPoena,
                   int potrebnoPoena, int potrebnaDuzinaSerije) {
  // analiziramo sve moguce pocetne utakmice
  int brojUtakmica = postigaoPoena.size();
  for (int i = 0; i + potrebnaDuzinaSerije <= brojUtakmica; i++) {
    // proveravamo da li na poziciji i pocinje serija potrebne duzine
    bool uSeriji = true;
    for (int j = i; j < i + potrebnaDuzinaSerije && uSeriji; j++)
      if (postigaoPoena[j] < potrebnoPoena)
        uSeriji = false;
    // ako pocinje, pronadjenja je serija sjajnih partija
    if (uSeriji)
      return true;
  }
  // nije pronadjena serija sjajnih partija
  return false;
}

int main() {
  // ucitavamo podatke
  int potrebnoPoena;
  cin >> potrebnoPoena;
  int potrebnaDuzinaSerije;
  cin >> potrebnaDuzinaSerije;
  int brojUtakmica;
  cin >> brojUtakmica;
  vector<int> postigaoPoena(brojUtakmica);
  for (int i = 0; i < brojUtakmica; i++)
    cin >> postigaoPoena[i];

  // proveravamo da li postoji serija
  bool postoji = postojiSerija(postigaoPoena,
                               potrebnoPoena, potrebnaDuzinaSerije);
  cout << (postoji ? "da" : "ne") << endl;

  return 0;
}
```

Одсецање непотребних провера

Прилично је јасно да није неопходно проверавати све серије дужине k. Можемо на исти начин као у задатку Најдужа серија победа рачунати дужину серије сјајних партија која почиње на позицији i (при чему i креће

од нуле). Дужину серије рачунамо инкрементално, проширујући је од позиције i надесно, све док су партије сјајне (ако су све партије у некој серији сјајне и ако се након тога одигра сјајна партија, све партије у проширеној серији су такође сјајне). Ако се та серија завршава на позицији j и ако је та серија дужа од потребне дужине k, успешно завршавамо поступак. У супротном знамо да су све њене подсерије (оне које крећу на позицијама између $i+1$ и j) такође краће од k, па анализу можемо наставити од позиције $i = j + 1$. Дакле, када се заврши нека серија сјајних партија која је краћа од k, проверу одмах треба наставити од позиције иза партије која је прекинула серију. Потребно је само бити пажљив и не заборавити проверу последње серије.

Пошто је довољан само један пролаз кроз серију, сложеност алгоритма је линеарна тј. $O(n)$. Елементи се не памте истовремено у меморији и меморијска сложеност је $O(1)$.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  int potrebnoPoena;
  cin >> potrebnoPoena;
  int potrebnaDuzinaSerije;
  cin >> potrebnaDuzinaSerije;
  int brojUtakmica;
  cin >> brojUtakmica;

  // da li je pronadjena serija sjajnih utakmica
  bool pronadjenaSerija = false;
  // duzina sjajnih utakmica
  int duzinaTekuceSerije = 0;
  for (int i = 0; i < brojUtakmica; i++) {
    // ucitavamo poene u utakmici na poziciji i
    int brojPoena;
    cin >> brojPoena;
    if (brojPoena >= potrebnoPoena) {
      // utakmica produzava seriju
      duzinaTekuceSerije++;
    } else {
      // zavrsena je tekuca serija i proveravamo da li je dovoljno dugacka
      if (duzinaTekuceSerije >= potrebnaDuzinaSerije)
        pronadjenaSerija = true;
      // nova serija jos nije zapoceta
      duzinaTekuceSerije = 0;
    }
  }
  // proveravamo da li je poslednja serija dovoljno dugacka
  if (duzinaTekuceSerije >= potrebnaDuzinaSerije)
    pronadjenaSerija = true;
  // ispisujemo rezultat
  cout << (pronadjenaSerija ? "da" : "ne") << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Најдужа серија победа

Кошаркашки тим је играо пуно утакмица у сезони. У свакој утакмици остварио је или победу или пораз. Напиши програм који одређује дужину најдуже серије победа у узастопним мечевима током сезоне.

Улаз: Са стандардног улаза се уноси природан број N ($5 \leq N \leq 50000$), а затим и N бројева -1 (што означава пораз) или 1 (што означава победу).

Излаз: На стандардни излаз исписати један природан број који представља тражену дужину најдуже серије узастопних победа.

Пример

Улаз	*Излаз*
8 1 1 -1 1 1 1 -1 -1	3

Задатак: Број растућих сегмената

Дат је низ a целих бројева, дужине n. Написати програм којим се одређује на колико начина можемо изабрати растуће сегменте у низу. Растући сегмент чине узастопни елементи низа $a_p < a_{p+1} < \ldots < a_q, 0 \leq p < q < n$.

Улаз: Прва линија стандардног улаза садржи природан број n ($2 \leq n \leq 10000$), број елемената низа. У свакој од n наредних линија стандардног улаза, налази по један члан низа.

Излаз: На стандардном излазу приказати у једној линији број растућих сегмената датог низа.

Пример

Улаз	*Излаз*
5 1 3 4 -2 10	4

Објашњење

То су низови [1, 3], [1, 3, 4], [3, 4], [-2, 10].

Види другачија решења овог задатка.

Задатак: Максимални збир сегмента

Напиши програм који одређује највећи збир неког сегмента (подниза узастопних елемената) датог низа.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50\,000$), а затим n целих бројева између -10 и 10, сваки број у посебном реду.

Излаз: На стандардни излаз испиши тражени збир.

Пример

Улаз	*Излаз*
6 2 -3 4 -1 3 -2	6

Решење

Груба сила

Најдиректнији могући начин да се задатак реши је да се израчуна збир сваког сегмента. Збир сваког сегмента можемо израчунавати засебно (у петљи или библиотечком функцијом), међутим, ефикасније решење

добијамо ако сегменте набрајамо редом (угнежђеним петљама, где спољашња петља набраја редом леве, а унутрашња десне крајеве сегмената) и збир наредног сегмента израчунавамо инкрементално, на основу збира претходног сегмента. Та техника је објашњена у задатку Префикс највећег збира.

Ако збир сваког сегмента рачунамо независно, сабирањем његових елемената (било у петљи, било помоћу библиотечке функције), сложеност решења је $O(n^3)$. Ако збирове сегмената рачунамо инкрементално, добијамо алгоритам сложености $O(n^2)$. У оба случаја елементе учитавамо у низ и меморијска сложеност је $O(n)$.

```
int max = 0;
for (int i = 0; i < n; i++) {
  int z = 0;
  for (int j = i; j < n; j++) {
    z += a[j];
    if (z > max)
      max = z;
  }
}
cout << max << endl;
```

Одсецање непотребних провера

Алгоритам заснован на провери свих сегмената се слободно може назвати тривијалним, јер се до њега долази прилично директно и веома једноставно му се и доказује коректност и анализира сложеност. Међутим, он је прилично неефикасан за решавање овог проблема, чак и када се збирови рачунају инкрементално. Значајно унапређење можемо добити када приметимо да велики број сегмената уопште не морамо да обрађујемо, јер из неких других разлога знамо да њихов збир не може бити максималан.

Посматрајмо низ `-2 3 2 -3 -3 4 -2 5 -8 3` и збирове свих његових непразних сегмената.

```
-2  1   3   0  -3   1  -1   4  -4  -1
    3   5   2  -1   3   1   6  -2   1
        2  -1  -4   0  -2   3  -5  -2
           -3  -6  -2  -4   1  -7  -4
               -3   1  -1   4  -4  -1
                    4   2   7  -1   2
                       -2   3  -5  -2
                            5  -3   0
                               -8  -5
                                    3
```

Прекид после негативног збира

Размотримо било који низ који почиње негативним бројем. Ниједан сегмент који почиње од тог броја, не може бити сегмент максималног збира, пошто се изостављањем првог броја добија већи збир. Ово својство је и општије. Уколико низ почиње префиксом негативног збира, из истог разлога, ниједан сегмент чији је он префикс не може бити сегмент максималног збира. Отуд, при инкременталном проширивању интервала удесно, чим се установи да је текући збир негативан, могуће је прекинути даље проширивање.

На пример, чим видимо да је први елемент првог сегмента `-2`, можемо прекинути даљу обраду елемената прве врсте, јер ће сви елементи друге врсте сигурно бити за два већи него одговарајући елементи прве врсте (`3` је веће од `1`, `5` је веће од `3`, `2` је веће од `0` итд.).

Слично, када се приликом проширивања сегмента који почиње на позицији `1` (од елемента `3`) дође до тога да је парцијални збир `-1` (што се дешава када се израчуна збир $3 + 2 - 3 - 3 = -1$, можемо прекинути са обрадом даљих сегмената који почињу на тој позицији, јер смо сигурни да ће за сваки од њих касније већи бити онај који се добија изостављање префикса `3 2 -3 -3`, чији је збир `-1`. Заиста од преосталих збирова `3 1 6 -2 1` у другој врсти за један су већи збирови `4 2 7 -1 2` у шестој врсти који су добијени изостављањем тог префикса. Обратимо пажњу на то да прекид унутрашње петље на овај начин узрокује да се максимална вредност у текућој врсти не мора уопште наћи. Петља која обрађује другу врсту ће бити прекинута чим се наиђе на збир `-1`, када је текућа вредност максимума `5` иако је максимум те врсте `6`. Сигурни смо да ће у некој наредној врсти постојати већа вредност од те највеће (заиста, у шестој врсти се јавља `7`), па нам налажење стварног максимума у текућој врсти уопште није неопходно.

Иако се на овај начин може прескочити разматрање неких сегмената, у најгорем случају сложеност није смањена. На пример, у случају да су елементи низа строго позитивни, збир никад не постаје негативан и сложеност након овог исецања је и даље квадратне сложености тј. $O(n^2)$.

Одсецање провере почетака унутар позитивног сегмента

Ако су сви елементи позитивни, максималан збир бива нађен за $i = 0$ и $j = n - 1$. Након тога се, увећавањем индекса i, збир смањује пошто се сваким скраћивањем сегмента слева изоставља неки позитиван број који доприноси збиру. И ово запажање се може уопштити. Не само што је непожељно скратити интервал слева за неки позитиван број, већ је непожељно скратити га за било који префикс чији је збир позитиван. Питање је докле такви префикси сежу? Бар до елемента чијим обухватањем добијамо први негативан префикс. Отуд сегмент максималног збира не може почињати ни на једној позицији између текуће почетне позиције и прве позиције на којој збир постаје негативан.

На пример, у наведеном примеру максимални сегмент не може почињати на позицији `2`, јер се проширивањем налево и додавањем елемента `3` са позиције `1` добијају сигурно збирови који су већи за три. Дакле, сви елементи друге врсте (која одговара позицији `1` у низу) су за `3` већи од одговарајућих елемената треће врсте (која одговара позицији `2` у низу). Заиста, `5` је веће од `2`, `2` од `-1` итд. Слично, ти елементи су за `5` већи од одговарајућих елемената четврте врсте (која одговара позицији `3` у низу). Заиста, `2` је веће од `-3`, `-1` од `-6` итд. Они су за `2` већи од одговарајући елемената пете врсте (која одговара позицији `4` у низу). Заиста, `-1` је веће од `-3`, `-3` је веће од `-5` итд. Зато те три врсте уопште нема потребе разматрати.

Захваљујући овом запажању, при завршетку обраде једне врсте и преласку на наредну, није неопходно увећавати променљиву i за један, већ је могуће наставити иза елемента чијим је укључивањем збир постао негативан.

Пошто се сваки елемент обрађује само једном, приликом имплементације није неопходно све елементе памтити у низу.

Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности промењљивих током обраде елемената низа.

```
i    j   max    z     a_j
0         0
     0          0
     0         -2    -2
1
     1          0
     1    3     3     3
     2    5     5     2
     3    5     2    -3
     4         -1    -3
5
     5          0
     5         -2    -2
6
     6          0
     6    5     4     4
     7    9     9     5
     8    9     1    -8
     9    9     4     3
    10
11
```

Пошто обе промењљиве пролазе кроз распон од 0 до n и крећу се само у једном смеру (вредност им се само повећава и никада не смањује), сложеност овог решења је линеарна тј. $O(n)$. У приказаној имплементацији елементи се чувају у низу па је и меморијска сложеност линеарна тј. $O(n)$, међутим, пошто се сваки елемент анализира само једном, за тим нема потребе и могуће је направити и имплементацију константне меморијске сложености.

```
int max = 0;
int i = 0;
while (i < n) {
```

```
  int z = 0;
  int j;
  for (j = i; j < n; j++) {
    z += a[j];
    if (z < 0)
      break;
    if (z > max)
      max = z;
  }
  i = j + 1;
}
cout << max << endl;
```

Види другачија решења овог задатка.

Задатак: Својство 132

Низ $a_0, a_1, \ldots, a_{n-1}$ задовољава 132-својство ако постоји тројка индекса $0 \leq i < j < k < n$, тако да је $a_i < a_k < a_j$. Напиши програм који испитује да ли низ задовољава 132-својство.

Улаз: Са стандардног улаза се учитава број елемената низа n ($3 \leq n \leq 10^5$), а затим и n елемената низа (раздвојених размаком).

Излаз: На стандардни излаз исписати `da` или `ne` у зависности од тога да ли низ задовољава 132-својство или не.

Пример 1

Улаз
```
4
1 2 3 4
```
Излаз
```
ne
```

Пример 2

Улаз
```
4
3 1 4 2
```
Излаз
```
da
```

Објашњење

На пример, елементи $3, 4, 2$ задовољавају 132-својство.

Пример 3

Улаз

```
7
9 11 8 9 10 7 9
```

Излаз

```
da
```

Објашњење

На пример, елементи $9, 11, 7$ задовољавају 132-својство.

Решење

Груба сила

Решење грубом силом подразумева да се провере све тројке елемената у низу.

Сложеност овог приступа одговара броју тројки $\binom{n}{3}$ и једнака је $O(n^3)$.

```
#include <iostream>
#include <vector>

using namespace std;

bool svojstvo132(const vector<int>& a) {
```

```
  int n = a.size();
  // proveravamo sve trojke pozicija i < j < k
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      for (int k = j+1; k < n; k++)
        if (a[i] < a[j] && a[i] < a[k] && a[j] > a[k])
          return true;
  return false;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << (svojstvo132(a) ? "da" : "ne") << endl;
  return 0;
}
```

Минимуми префикса

За сваки број a_j у низу проверићемо да ли може бити средишњи елемент 132-тројке. За то је потребно да лево од a_j постоји елемент a_i, а десно од a_j елемент a_k, тако да су оба строго мања од a_j и да је $a_i < a_k$.

Ако лево од a_j не постоји елемент строго мањи од a_j, елемент a_j не може бити средишњи елемент 132-тројке. Ако лево од a_j постоји више елемената a_i који су строго мањи од a_j, довољно је да размотримо само најмањи од њих (јер се тиме повећава шанса да десно од a_j постоји елемент мањи од a_j, а већи од a_i). Наиме, ако постоји нека 132-тројка (a_i, a_j, a_k) и ако је a'_i елемент лево од a_j, који је мањи од a_i, тада је (a'_i, a_j, a_k) такође 132-тројка елемената. Дакле, уместо разматрања свих парова (a_i, a_j), за $i < j$, можемо извршити одсецање и посматрати само онај пар у коме је a_i минимално од свих елемената лево од a_j строго мањих од a_j, што значајно убрзава претрагу.

За дати елемент a_j, елемент a_i можемо одредити као минимум префикса низа закључно са елементом a_j (префикса $a_0, \ldots, a_j$). Минимуме префикса можемо одржавати инкрементално (као у задатку Најбољи "сабмит"). Текући елемент a_j може бити средишњи елемент 132-тројке ако и само ако десно од њега постоји неки елемент који је строго између a_i и a_j тј. припада интервалу (a_i, a_j). Ако је $a_i = a_j$, тада је интервал (a_i, a_j) празан и a_j не може средишњи елемент 132-тројке. У супротном, можемо обилазити све елементе a_k десно од a_j и линеарном претрагом проверавати да ли припадају том интервалу.

За сваки елемент a_j обилазимо све елементе десно од њега, па је сложеност овог решења $O(n^2)$.

```
#include <iostream>
#include <vector>

using namespace std;

bool svojstvo132(const vector<int>& a) {
  int n = a.size();
  // minimum prefiksa na pozicijama [0, j]
  int minP = a[0];
  // za svaki element aj proveravamo da li moze biti sredisnji u nekoj
  // 132-trojci
  for (int j = 1; j < n-1; j++) {
    // azuriramo minimum prefiksa na pozicijama [0, j]
    minP = min(minP, a[j]);
    // analiziramo interval (ai, aj) i proveravamo da li postoji ak
    // desno od j koji mu pripada
    int ai = minP, aj = a[j];
    // interval je prazan
    if (ai == aj) continue;
```

```
    // analiziramo sve elemente ak desno od aj
    for (int k = j+1; k < n; k++)
      if (ai < a[k] && a[k] < aj)
        // pronadjena je 132-trojka (ai, aj, ak)
        return true;
  }
  // ni jedna trojka nije pronadjena
  return false;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << (svojstvo132(a) ? "da" : "ne") << endl;
  return 0;
}
```

Види другачија решења овог задатка.

2.8 Сортирање

Због својих примена, сортирање је један од најзначајнијих и најпроучаванијих проблема у рачунарству. Током година развијен је велики број алгоритама за сортирање. Најпознатији су *QuickSort*, *MergeSort*, *HeapSort*, *SelectionSort*, *InsertionSort*, *BubbleSort* и *ShellSort*. Неки од њих су једноставнији, али спорије раде (такви су *Selection*, *Insertion* и *Bubble*), док су неки ефикаснији, али компликованији за разумевање и реализацију (такви су *Quick*, *Merge*, *Heap* и *Shell*). Њихова ручна имплементација поучна је као одличан пример за приказ различитих техника конструкције алгоритама.

Ипак, по правилу све библиотеке савремених (а и старијих) програмских језика нуде готове функције за сортирање низова. Коришћење ових функција је увек пожељнија опција у односу на ручну имплементацију (добијају се краћи и разумљивији програми, функције су пажљиво тестиране и скоро извесно потпуно коректне, и имплементиране су на најефикаснији могући начин). Можемо слободно претпоставити да је сложеност и најгорег и просечног случаја код библиотечких функција сортирања квазилинеарна тј. једнака $O(n \log n)$ - у делу посвећеном рекурзији и техници "подели па владај", биће приказани најзначајнији алгоритми сортирања (који се користе и у библиотечким функцијама сортирања) и њихова анализа сложености.

У наставку овог поглавља кроз неколико веома елементарних примера ћемо приказати употребу библиотечких функција сортирања, као и пар елементарних алгоритама сортирања (који се једноставно имплементирају, али су квадратне сложености и стога их никада у пракси не треба употребљавати).

Сортирање подразумева да се елементи ређају у односу на неки поредак. Бројеви се подразумевано ређају по њиховој нумеричкој вредности, ниске се подразумевано ређају лексикографски абецедно (као у речнику), парови и торке се поново ређају лексикографски, по својим компонентама, међутим, често је потребно сортирати низове елемената над којима није дефинисан подразумевани поредак (на пример, низове структура). Библиотечке функције сортирања допуштају навођење различитих критеријума сортирања (задавањем функција поређења), што ће такође бити илустровано кроз неколико елементарних задатака.

Након тога, приказаћемо употребу сортирања као технике претпроцесирања која омогућава да се снизи сложеност решења задатака. Сортирање је толико корисна техника, да у свету конструкције алгоритама често важи правило "Ако не знаш одакле да кренеш да конструишеш алгоритам, ти прво покушај да сортираш податке". Једна од најзначајнијих примена сортирања је то што се сортирани подаци могу брзо претраживати (применом алгоритма бинарне претраге). Стога ће у наредном поглављу посебна пажња бити посвећена управо томе.

2.8.1 Функције сортирања

У језику C++ сортирање се врши библиотечком функцијом `sort`. Функцији се прослеђују два итератора (или два показивача) који ограничавају распон који се сортира. На пример, вектор `a` се може сортирати помоћу

`sort(a.begin(), a.end())` или `sort(begin(a), end(a))`, док се низ `a` дужине `n` може сортирати помоћу `sort(a, a+n)`.

Гарантована сложеност најгорег случаја функције `sort` је $O(n \log n)$ где је n број елемената који се сортирају.

Задатак: Сортирање бројева

Напиши програм који уређује (сортира) низ бројева неопадајуће (сваки наредни мора да буде већи или једнак од претходног).

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 5 \cdot 10^4$) а затим и n природних бројева мањих од $2n$, сваки у посебном реду.

Излаз: На стандардни излаз исписати учитане бројеве у сортираном редоследу.

Пример

Улаз	*Излаз*
5	1
3	1
1	3
6	6
8	8
1	

Решење

Сортирање вектора библиотечком функцијом

У језику C++ сортирање је најбоље вршити функцијом `sort`, која прима два итератора - први који указује на почетак дела низа тј. вектора који се сортира, а други указује иза последњег елемента у делу низа који се сортира. Када се сортира цео вектор `a` ти итератори се могу добити помоћу `a.begin()` и `a.end()`.

Сложеност овог приступа је $O(n \log n)$.

```
// ucitavanje brojeva
int n;
cin >> n;
vector<int> a(n);
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiranje
sort(a.begin(), a.end());

// ispis rezultata
for (int i = 0; i < n; i++)
  cout << a[i] << endl;
```

Сортирање низа библиотечком функцијом

Када се сортира низ `a` са `n` елемената почетни итератор је `a`, а завршни се може добити као `next(a, n)` или евентуално `a+n`.

Сложеност овог приступа је $O(n \log n)$.

```
// ucitavanje brojeva
int n;
cin >> n;
int a[MAX];
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiranje
sort(a, next(a, n));
```

```
// ispis rezultata
for (int i = 0; i < n; i++)
  cout << a[i] << endl;
```

Сортирање селекцијом (SelectionSort)

Алгоритам сортирања селекцијом (*SelectionSort*) низ сортира тако што се у првом кораку најмањи елемент доводи на прво место, у наредном кораку се најмањи од преосталих елемената доводи на друго место и тако редом док се низ не сортира.

Најједноставнији начин да се алгоритам реализује је да се у спољној петљи разматрају редом позиције i у низу од прве до претпоследње, да се у унутрашњој петљи разматрају све позиције j од $i+1$ па до последње и да се у телу унутрашње петље размењују елементи на позицији i и j, ако су тренутно у наопаком редоследу.

У најгорем случају и број поређења и број размена је $O(n^2)$.

```
void selection_sort(vector<int>& a) {
  for (int i = 0; i < a.size(); i++) {
    for (int j = i + 1; j < a.size(); j++)
      if (a[i] > a[j])
        swap(a[i], a[j]);
  }
}
```

Бољи начин имплементације је да се за сваку позицију i прво пронађе позиција најмањег елемента у делу низа од позиције i до краја, и да се онда елемент на позицији i размени са пронађеним минимумом (тиме се број размена значајно смањује). Позицију најмањег елемента проналазимо на исти начин као у задатку Редни број максимума.

Нагласимо да је чак и са овом бољом имплементацијом овај алгоритам прилично неефикасан и није га пожељно примењивати осим у случају релативно кратких низова (мада се и они сортирају једноставније библиотечком функцијом).

Алгоритам врши увек $O(n)$ размена и $O(n^2)$ поређења, па је сложеност квадратна.

```
void selection_sort(vector<int>& a) {
  // na svaku poziciju i dovodimo najmanji element iz dela niza na
  // pozicijama [i, n)
  for (int i = 0; i < a.size(); i++) {
    // pozicija najmanjeg elementa u delu [i, n)
    int pmin = i;
    for (int j = i + 1; j < a.size(); j++)
      // ako je element na poziciji j manji od elementa na poziciji
      // pmin, nasli smo novi minimum, pa azuriramo vrednost pmin
      if (a[pmin] > a[j])
        pmin = j;
    // razmenjujemo element na poziciji i sa minimumom
    swap(a[i], a[pmin]);
  }
}
```

Сортирањем уметањем (InsertionSort)

Алгоритам сортирања уметањем (*InsertionSort*) заснива се на томе да се сваки наредни елемент у низу умеће на своје место у сортираном префиксу низа испред њега.

Најједоставнија имплементација је таква да се у спољној петљи разматрају све позиције од друге, па до краја низа и да се уметање врши тако што се сваки елемент размењује са елементима испред себе, све док се испред њега не појави елемент који није мањи од њега или док елемент не стигне на почетак низа.

У најгорем случају је и број поређења и број размена квадратни тј. $O(n^2)$.

```
void insertion_sort(vector<int>& a) {
  // element sa pozicije i umecemo u vec sortirani prefiks na
  // pozicijama [0, i)
```

```
  for (int i = 1; i < a.size(); i++) {
    // dok je element na poziciji j manji od njemu prethodnog
    // razmenjujemo ih
    for (int j = i; j > 0 && a[j] < a[j-1]; j--)
      swap(a[j], a[j-1]);
  }
}
```

У бољој имплементацији избегавају се размене током уметања. Елемент који се умеће се памти у привременој променљивој, затим се сви елементи мањи од њега померају за једно место удесно и на крају се запамћени елемент уписује на своје место.

У овом алгоритму се не врше размене, него појединачне доделе. Размена захтева три доделе, па се овом оптимизацијом фаза довођења елемента на своје место може убрзати око три пута. Међутим, у најгорем случају је и број поређења и број додела квадратни тј. $O(n^2)$.

```
void insertion_sort(vector<int>& a) {
  // element sa pozicije i umecemo u vec sortirani prefiks na
  // pozicijama [0, i)
  for (int i = 1; i < a.size(); i++) {
    // pamtimo element na poziciji i
    int tmp = a[i];
    // sve elemente koji su veci od njega pomeramo za
    // jednu poziciju udesno
    int j;
    for (j = i - 1; j >= 0 && a[j] > tmp; j--)
      a[j + 1] =  a[j];
    // stavljamo element sa pozicije i na njegovo mesto
    a[j + 1] = tmp;
  }
}
```

Види другачија решења овог задатка.

Задатак: Медијана

Постоји неколико мера које одређују средину дате серије бројева. Најпознатија је аритметичка средина тј. просек, међутим, она је прилично осетљива на грешке у подацима и неколико елемената који значајно одступају од осталих могу прилично да измене просек. На пример, просек бројева 3, 2, 1, 5, 4 је 3, међутим, ако им се дода број 99, тада просек постаје око 19, што је јако велика промена настала под утицајем само једног елемента, који може бити и резултат неке грешке у подацима. Зато се често разматра медијана која се добија тако што се низ сортира и посматра се средишњи елемент (ако је укупан број елемената непаран), тј. просек два средишња елемента (ако је укупан број елемената паран). На пример, ако се наша полазна серија сортира добија се 1, 2, 3, 4, 5 и ту је средишњи елемент 3 и он је уједно и медијана, а ако се придода и 99, тада су два средишња елемента 3 и 4 и медијана је 3,5. Напиши функцију која одређује медијану дате серије бројева.

Израчунавање медијане ћемо тестирати тако што ћемо га применити на серију бројева која се добија тако што се примени рекурентна формула $a_{i+1} = c_0 \cdot a_i + c_1$, почевши од задатог елемента a_0. На пример, ако је $a_0 = 0$, $c_0 = 4$ и $c_1 = 1$, добија се серија $0, 1, 5, 21, 85, \ldots$. При том се сва аритметика изводи са неозначеним бројевима, по модулу 2^{32}.

Улаз: У првој линији стандардног улаза налази се број n који представља број елемената серије чију медијану треба израчунати. У другој бројеви c_0, c_1, раздвојени са по једним размаком. У трећој линији се налази се број a_0.

Излаз: На стандардни излаз исписати вредност медијане заокружене на 2 децимале.

Пример 1

Улаз	*Излаз*
5	5
4 1	
0	

Пример 2

Улаз	*Излаз*
10 1664525 1013904223 1	1586537357.00

Објашњење

Серија бројева која се анализира је

```
1
1015568748
1586005467
2165703038
3027450565
217083232
1587069247
3327581586
2388811721
70837908
```

Када се сортира, добија се серија

```
1
70837908
217083232
1015568748
1586005467
1587069247
2165703038
2388811721
3027450565
3327581586
```

Медијана је аритметичка средина између средишња два елемента $(1586005467 + 1587069247)/2 = 1586537357$.

Решење

Сортирање

Директан начин да се пронађе медијана је да се претходно сортира низ. Када је низ сортиран, ако је број елемената непаран потребно је вратити елемент на позицији `n/2`, а ако је збир паран онда аритметичку средину елемената на позицијама `n/2` и `n/2 + 1` (водећи рачуна да приликом њеног рачунања не дође до прекорачења).

Ако се употреби библиотечка функција сортирања (као у задатку Сортирање бројева), сортирање се обавља у времену $O(n \log n)$, што је и укупна сложеност овог приступа.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>

using namespace std;

double medijana(vector<unsigned>& a) {
  sort(a.begin(), a.end());
  if (a.size() % 2 != 0)
    return a[a.size() / 2];
  else
    return ((double)a[a.size() / 2 - 1] + (double)a[a.size() / 2]) / 2.0;
}

int main() {
  int n;
```

```
  cin >> n;
  unsigned c0, c1;
  cin >> c0 >> c1;
  vector<unsigned> a(n);
  cin >> a[0];
  for (int i = 1; i < n; i++)
    a[i] = c0 * a[i-1] + c1;
  cout << fixed << showpoint << setprecision(2) << medijana(a) << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Цик цак селекција

Уредити низ бројева тако да најмањи од свих елемената иде на прво место, најмањи од преосталих на последње, следећи најмањи по величини на друго место, следећи на претпоследње итд.

Улаз: У првој линији стандардног улаза унети број елемената низа n ($1 \leq n \leq 10^5$), а затим у следећих n линија елементе низа $-10^6 \leq a_i \leq 10^6$.

Излаз: У свакој од n излазних линија приказати по један елемент на тражени начин уређеног низа.

Пример

Улаз	*Излаз*
7	-5
-3	1
7	7
1	9
8	8
-5	2
2	-3
9	

Решење

Модификација сортирања селекцијом

Једно могуће решење се заснива на модификацији алгоритма сортирања селекцијом (његову смо имплементацију приказали у задатку Сортирање бројева). Нека је полазни низ $a_0, a_1, \ldots, a_{n-1}$. Поново ћемо одржавати индекс прве слободне позиције са леве i и индекс прве слободне позиције са десне стране низа j (иницијализоваћемо их са 0 и $n-1$). Након тога, понавља се следеће: најмањи елемент сегмента $a_i, a_{i+1}, \ldots,$ разменићемо са a_i и увећати индекс почетка сегмента i за 1; за нови сегмент (претходни сегмент без првог елемента сегмента) разменићемо последњи елемент сегмента a са најмањим и умањити индекс краја сегмента j за 1. Претходни поступак ћемо понављати све док се док се не сусретну индекс левог и индекс десног краја сегмента (док је сегмент одређен позицијама i и j непразан тј. док је $i < j$), а контролу да ли текући минимум треба ставити на почетак или крај можемо остварити логичком променљивом којој мењамо вредност у сваком кораку.

Сложеност овог приступа одговара сложености сортирања селекцијом, која износи $O(n^2)$, па је ово решење недопустиво неефикасно.

```
#include <iostream>
#include <algorithm>
using namespace std;

const int N_MAX = (int)(1e5);

// vraca indeks minimalnog elementa podniza a[i],...,a[j]
int indeksMinimuma(int a[], int i, int j) {
  int ind = i;
  for (int k = i + 1; k <= j; k++)
    if (a[k] < a[ind])
```

```
      ind = k;
  return ind;
}

// soritra niz tako da prvi po velicini (najmanji) ide na mesto 0,
// drugi na mesto n-1, treci na mesto 1, cetvrti n-2 itd.
void sortirajCikCak(int a[], int n) {
  // odredjuje da li naredni element po velicini ide na pocetak ili na kraj
  bool naPocetak = true;
  // pozicije na koje u nizu postavljamo elemente sa leve strane i sa
  // desne strane
  int i = 0, j = n-1;
  while (i < j) {
    // trazimo poziciju najmanjeg elementa u delu [i, j]
    int indMin = indeksMinimuma(a, i, j);
    if (naPocetak)
      // minimum stavljamo na pocetak
      swap(a[indMin], a[i++]);
    else
      // minimum stavljamo na kraj
      swap(a[indMin], a[j--]);
    // sledeci element ide na suprotan kraj niza
    naPocetak = !naPocetak;
  }
}

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  int a[N_MAX];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // sortiramo ga cik-cak
  sortirajCikCak(a, n);

  // ispisujemo rezultat
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;
}
```

Класично сортирање и помоћни низ

Ако допустимо коришћење помоћног низа за смештање резултата, једно решење је да сортирамо улазни низ у неопадајући поредак (најбоље библиотечком функцкијом, слично као у задатку Сортирање бројева), а затим све елементе тако сортираног низа копирати у тај помоћни низ и то тако што све елементе са парних позиција копирамо на почетак, а све елементе са непарних позиција на крај помоћног низа чији садржај на крају исписујемо (при том је потребно да у две променљиве чувамо прво слободно место са почетка и прво слободно место са краја на које уписујемо елементе - сличну технику видели смо, на пример, у задатку Обртање низа).

Сложеношћу овог приступа доминира сортирање, па је укупна сложеност $O(n \log n)$. Иако се користи додатни низ, меморијска сложеност остаје $O(n)$.

```
#include <iostream>
#include <algorithm>
using namespace std;
```

```
// maksimalni broj elemenata niza odredjen uslovima zadatka
const int N_MAX = (int)1e5;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo elemente niza
  int n;
  cin >> n;
  int a[N_MAX];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // sortiramo ceo niz neopadajuce
  sort(a, a + n);

  // drugi niz u koji cemo upisati resenje
  int b[N_MAX];
  // pozicije na koje u nizu b postavljamo elemente sa leve strane i
  // sa desne strane
  int l = 0, d = n-1;
  // obradjujemo sve elemente niza a
  for (int i = 0; i < n; i++) {
    if (i % 2 == 0)
      // one sa parnih pozicija stavljamo na pocetak niza b
      b[l++] = a[i];
    else
      // a one sa neparnih pozicija stavljamo na kraj niza b
      b[d--] = a[i];
  }

  // stampamo sadrzaj niza b
  for (int i = 0; i < n; i++)
    cout << b[i] << endl;
}
```

2.8.2 Сортирање по разним критеријумима

Као што је речено, за многе типове података (бројеви, ниске, торке) постоји подразумевани поредак, који се користи када при сортирању не прецизирамо критеријум.

У случају да нам подразумевани поредак не одговара или да за податке које сортирамо подразумевани поредак није дефинисан, у библиотечким функцијама сортирања можемо да задамо критеријум сортирања какав год нам одговара.

Често је једина промена која се захтева у односу на подразумевани поредак то да низ буде сортиран нерастуће уместо неопадајуће. Један начин да се то постигне је да се изврши обртање низа након сортирања. Тај додатни посао се може избећи ако се као додатни параметар функцији сортирања проследи `greater<...>`. На пример, вектор бројева `a` можемо сортирати опадајуће, позивом `sort(begin(a), end(a), greater<int>())`.

Разни начини да се библиотечкој функцији сортирања зада специфичан критеријум поређења описани су у задатку Сортирање такмичара.

Када се сортира низ структура или објеката, критеријум је могуће задати тако што се у самој структури тј. класи дефинише метода која врши поређење. У језику C++ потребно је дефинисати оператор `<`. Та метода треба да врати `true` ако и само ако је објекат на ком је позвана строго мањи од објекта који јој је прослеђен као аргумент (мањи објекти иду испред већих у сортираном редоследу).

Приликом сортирања било ког типа елемената, функцији за сортирање могуће је као додатни параметар проследити функцију поређења (било анонимну, било именовану). Она прима два објекта која треба да упореди

и враћа `true` ако и само ако јој је први прослеђен аргумент строго мањи од од другог (мањи објекти иду испред већих у сортираном редоследу).

Библиотечка функција сортирања гарантује да ће број поређења и размена елемената бити $O(n \log n)$, где је n број елемената дела низа који се сортира. Ако је поређење константне сложености, то гарантује да ће сложеност сортирања бити $O(n \log n)$. Међутим, ако је функција поређења кориснички дефинисана, она може бити и веће сложености од константне, што значајно може успорити процес сортирања.

Задатак: Сортирање такмичара

Дат је низ такмичара, за сваког такмичара познато је његово име и број поена на такмичењу. Написати програм којим се сортира низ такмичара нерастуће по броју поена, а ако два такмичара имају исти број поена, онда их уредити по имену у неопадајућем поретку.

Улаз: У првој линији стандардног улаза налази се природан број n ($n \leq 50000$). У следећих n линија налазе се редом елементи низа. За сваког такмичара, у једној линији, налази се одвојени једним бланко симболом, његово име (дужине највише 20 карактера) и број поена (природан број из интервала $[0, 10000]$) које је такмичар освојио.

Излаз: На стандардни излаз исписати елементе уређеног низа такмичара, за сваког такмичара у једној линији приказати његово име и број поена, одвојени једним бланко симболом.

Пример

Улаз	*Излаз*
`5`	`Milica 89`
`Maja 56`	`Jovan 78`
`Marko 78`	`Marko 78`
`Krsto 23`	`Maja 56`
`Jovan 78`	`Krsto 23`
`Milica 89`	

Решење

Прво питање које је потребно разрешити је како чувати податке о такмичарима. Најприродније решење да се подаци о такмичару памте у структури `Takmicar` чији су елементи име такмичара (податак типа `string`) и број поена (податак типа `int`). За чување информација о свим такмичарима можемо онда употребити неку колекцију структура. Друге могућности су да се подаци чувају у два низа (низу имена и низу поена) или да се уместо структура користе парови тј. торке.

Сортирање је могуће извршити на разне начине (види задатак Сортирање бројева).

Сортирање селекцијом

Један од начина је да се ручно имплементира неки алгоритам сортирања (као пример, наводимо најједноставнију имплементацију алгоритма SelectionSort), међутим то је обично веома компликовано и неефикасно.

Сложеност сортирања селекцијом је $O(n^2)$.

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

struct Takmicar {
    string ime;
    int brojPoena;
};

bool uporedi(Takmicar A, Takmicar B) {
  if (A.brojPoena > B.brojPoena)
    return true;
  if (A.brojPoena < B.brojPoena)
    return false;
  return A.ime <= B.ime;
```

```
}

int main() {
  int n;
  cin >> n;

  Takmicar a[50000];
  for (int i = 0; i < n; i++)
    cin >> a[i].ime >> a[i].brojPoena;

  for (int i = 0; i < n - 1; i++) {
    int iMax = i;
    for (int j = i + 1; j < n; j++)
      if (uporedi(a[j], a[iMax]))
        iMax = j;
    swap(a[iMax], a[i]);
  }

  for (int i = 0; i < n; i++)
    cout << a[i].ime << " " << a[i].brojPoena << endl;

  return 0;
}
```

Библиотечка функција сортирања

Најбољи и уједно и најједноставнији начин је употребити библиотечку функцију сортирања. У језику C++ то је функција `sort` (без обзира на то да ли се користи вектор или низ).

Све варијанте коришћења библиотечке функције у којој се поређење и размештање два елемента може извршити у константној сложености су квази-линеарне сложености $O(n \log n)$.

Функцији сортирања је потребно доставити и функцију поређења којом се заправо одређује редослед елемената након сортирања. Тело те функције поређења врши вишекритеријумско (лексикографско) поређење уређених двојки података, слично оном које смо видели у задатку Пунолетство. Прво се пореди број поена и ако је број поена првог такмичара мањи функција поређења враћа `false` (јер он треба да иде иза другог такмичара), ако је већи враћа се `true` (јер он треба да иде испред другог такмичара), а ако су бројеви поена једнаки, прелази се на поређење имена. За то се може упоредити библиотечко абецедно (лексикографско) поређење ниски (слично као у задатку Лексикографски минимум). У језику C++ за то се могу употребити оператори поређења (`<`, `>`, `<=`, `>=`, `==`, `!=`).

У језику C++ постоји неколико начина да дефинишемо функцију поређења.

Именована функција поређења

Први је да се дефинише класична (глобална) функција која прима две структуре (податке о два такмичара) и која врача вредност типа `bool` и то вредност `true` ако први такмичар треба да буде испред другог у коначном сортираном низу, тј. вредност `false` у супротном. Ефикасности ради, структуре које се пореде има смисла пренети преко константних референци. Та функција добија своје име (на пример, `poredi`) и то име се онда наводи као трећи параметар у позиву библиотечке функције `sort` (након два итератора који одређују распон који се сортира).

```
bool uporedi(const Takmicar& a, const Takmicar& b) {
  ...
}

sort(begin(a), end(a), uporedi);
```

Комплетно решење се може имплементирати на следећи начин.

```
#include <iostream>
#include <algorithm>
#include <string>
```

```
using namespace std;

struct Takmicar {
    string ime;
    int brojPoena;
};

bool uporedi(const Takmicar& A, const Takmicar& B) {
  if (A.brojPoena > B.brojPoena)
    return true;
  if (A.brojPoena < B.brojPoena)
    return false;
  return A.ime <= B.ime;
}

int main() {
  int n;
  cin >> n;

  Takmicar a[50000];
  for (int i = 0; i < n; i++)
    cin >> a[i].ime >> a[i].brojPoena;

  sort(a, a+n, uporedi);

  for (int i = 0; i < n; i++)
    cout << a[i].ime << " " << a[i].brojPoena << endl;
  return 0;
}
```

Анонимна функција поређења

Други начин је да се уместо именоване функције, функција поређења дефинише као анонимна функција тј. као λ-израз. Параметри и повратна вредност те функције су исти као у случају именоване функције, при чему се повратни тип не мора експлицитно навести. Пошто функција поређења у овом случају не мора да користи друге податке осим података о такмичарима који се пореде, као параметре затворења те функције (параметри који се наводе у заградама `[]`) нема потребе наводити ништа. Анонимна функција се наводи директно у позиву функције `sort` као њен трећи аргумент.

```
sort(a.begin(), a.end(),
     [](const Takmicar& a, const Takmicar& b) {
        ...
     });
```

Комплетно решење се може имплементирати на следећи начин.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

struct Takmicar {
    string ime;
    int brojPoena;
};

int main() {
  int n;
  cin >> n;
```

```
  vector<Takmicar> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i].ime >> a[i].brojPoena;

  sort(a.begin(), a.end(),
       [] (const Takmicar& A, const Takmicar& B) {
         if (A.brojPoena > B.brojPoena)
           return true;
         if (A.brojPoena < B.brojPoena)
           return false;
         return A.ime <= B.ime;
       });

  for (int i = 0; i < n; i++)
    cout << a[i].ime << " " << a[i].brojPoena << endl;

  return 0;
}
```

Објекат упоређивач

Трећи начин је да се дефинише структура (или класа) која имплементира функционалност поређења два такмичара у оквиру своје методе `operator()`. Парамтери и повратна вредност те функције је иста као и у претходним случајевима, а у позиву функције `sort` наводи се један објекат те структуре (или класе).

```
struct PoredjenjeTakmicara {
  bool operator() (const Takmicar& a, const Takmicar& b) {
     ...
  }
}

sort(a.begin(), a.end(), PoredjenjeTakmicara());
```

Природно је да такав објекат "упоређивач" буде објекат посебне структуре или класе, а могуће је додати га и некој већ постојећој структури (на пример, структури `Takmicar`). Комплетно решење се може имплементирати на следећи начин.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

struct Takmicar {
  string ime;
  int brojPoena;
};

struct PoredjenjeTakmicara {
  bool operator() (const Takmicar& A, const Takmicar& B) {
    return A.brojPoena > B.brojPoena ||
           (A.brojPoena == B.brojPoena && A.ime <= B.ime);
  }
};

int main() {
  int n;
  cin >> n;

  vector<Takmicar> a(n);
  for (int i = 0; i < n; i++)
```

```
    cin >> a[i].ime >> a[i].brojPoena;

  sort(a.begin(), a.end(), PoredjenjeTakmicara());

  for (int i = 0; i < n; i++)
    cout << a[i].ime << " " << a[i].brojPoena << endl;

  return 0;
}
```

Оператор поређења

Четврти начин је да се у самој структури која чува податке о такмичару дефинише метода под именом `operator<` којом се обезбеђује да се структуре `A` и `B` могу поредити помоћу `A < B`. Та метода има један параметар који представља другог такмичара, док је први такмичар она структура на којој се тај метод позива.

```
struct Takmicar {
    string ime;
    int brojPoena;

  bool operator<(const Takmicar& A) {
     ...
  }
};

sort(a.begin(), a.end());
```

Комплетно решење се може имплементирати на следећи начин.

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

struct Takmicar {
    string ime;
    int brojPoena;

  bool operator<(const Takmicar& A) {
    if (brojPoena > A.brojPoena)
      return true;
    if (brojPoena < A.brojPoena)
      return false;
    return ime <= A.ime;
  }
};

int main() {
  int n;
  cin >> n;

  Takmicar a[50000];
  for (int i = 0; i < n; i++)
    cin >> a[i].ime >> a[i].brojPoena;

  sort(a, a+n);

  for (int i = 0; i < n; i++)
    cout << a[i].ime << " " << a[i].brojPoena << endl;
  return 0;
}
```

Коришћење парова или торки

Рецимо и да се за репрезентацију података могу користити парови тј. торке (слично као у задатку Пунолетство). У језику C++ парови су представљени типом `pair`, а торке типом `tuple`. Пошто се парови тј. торке подразумевано пореде лексикографски (прво прва компонента, а тек ако је прва компонента једнака, онда друга) и неопадајуће, функцију поређења није неопходно наводити. Зато је паметно парове организовати тако да се као прва компонента памти супротан број од броја освојених поена (да би се добио опадајући редослед броја поена), а као друга име такмичара.

```
#include <iostream>
#include <algorithm>
#include <string>
#include <utility>
#include <vector>
using namespace std;

int main() {
  int n;
  cin >> n;

  vector<pair<int, string>> a(n);
  for (int i = 0; i < n; i++) {
    string ime; int brojPoena;
    cin >> ime >> brojPoena;
    a[i] = make_pair(-brojPoena, ime);
  }

  sort(a.begin(), a.end());

  for (int i = 0; i < n; i++)
    cout << a[i].second << " " << -a[i].first << endl;

  return 0;
}
```

Задатак: Сортирање на основи растојања од О

Пера је проучавао градове које жели да посети. На мапи је нацртао координатни систем у којем се његов град налази у координатном почетку, а остали градови су одређени њиховим Декартовим координатама. Написати програм којим се низ градова сортира у неопадајућем пореку на основу удаљености од Периног града (од најближег до најдаљег). Ако су два града подједнако удаљена од Периног, уредити их у нерастућем поретку прво по координати x па по координати y.

Улаз: У првој линији стандардног улаза налази се број градова, природан број n ($n \leq 50000$). У следећих n линија налазе се елементи низа: у свакој линији, налазе се, x и y координата једног града (два цела броја из интервала $[-10^3, 10^3]$ одвојена једним размаком).

Излаз: Стандардни излаз садржи елементе уређеног низа градова - за сваки град у једној линији приказати редом, његову координату x и y, раздвојене размаком.

Пример

Улаз	*Излаз*
4	1 -1
8 2	-2 -3
1 -1	2 -3
2 -3	8 2
-2 -3	

Решење

Сортирање низа структура

Овај је задатак веома сличан задатку Сортирање такмичара. Поново имамо низ слогова (у овом случају

градова представљених својим Декартовим координатама) и тражимо да се тај низ сортира на основу неког лексикографског поретка. Самим тим и решење може бити веома слично. Један начин је да градове представимо структурама, а да низ структура сортирамо коришћењем библиотечких функција.

Функција поређења прво треба да израчуна растојање сваке тачке од координатног почетка. Начин да се то уради је да се примени Питагорина теорема, као у задатку Растојање тачака. Ипак, да бисмо избегли баратање са реалним типом (јер без обзира што су координате реалне, растојање захтева израчунавање корена и може бити реалан број), можемо приметити да је корен монотона функција (корен је већи ако и само ако је већа поткорена величина) и уместо поређења растојања две тачке, можемо поредити квадрате тих растојања (израчунате помоћу израза $x^2 + y^2$). Ако су квадрати растојања једнаки, пореде се координате x, а ако су и оне једнаке пореде се координате y.

```
#include <iostream>
#include <algorithm>
using namespace std;

// struktura za predstavljanje tacke
struct Tacka {
  int x, y;
};

// kvadrat rastojanja tacke M od koordinatnog pocetka O
int KvadratRastojanjaOdO(Tacka M) {
  return M.x * M.x + M.y * M.y;
}

// funkcija koja poredi da li je tacka A manja od tacke B u definisanom poretku
bool uporedi(Tacka A, Tacka B) {
  int dA = KvadratRastojanjaOdO(A),
      dB = KvadratRastojanjaOdO(B);
  return dA < dB ||
         (dA == dB && A.x > B.x) ||
         (dA == dB && A.x == B.x && A.y > B.y);
}

int main() {
  // broj tacaka
  int n;
  cin >> n;

  // ucitavamo tacke u niz
  Tacka A[50000];
  for (int i = 0; i < n; i++)
    cin >> A[i].x >> A[i].y;

  // sortiramo tacke
  sort(A, A + n, uporedi);

  // ispisujemo tacke
  for (int i = 0; i < n; i++)
    cout << A[i].x << " " << A[i].y << endl;

  return 0;
}
```

Сортирање низа торки

Други начин је да се уместо структура употребе торке (`tuple` у језику C++) и чињеница да се оне подразумевано пореде лексикографски. За сваку учитану тачку (тј. град) можемо формирати тројку коју сачињава њен квадрат растојања од координатног почетка, затим супротна вредност координати x и супротна вредност

координате y (да би се добио нерастући поредак по тим координатама).

```
#include <iostream>
#include <vector>
#include <tuple>
#include <algorithm>
using namespace std;

int main() {
  // ucitavamo broj tacaka
  int n;
  cin >> n;
  // cuvamo tacke u nizu trojki koje sadrze kvadrat njihovog
  // rastojanja od 0, negiranu x-koordinatu i negiranu y-koordinatu
  vector<tuple<int, int, int>> A(n);
  // ucitavamo tacke
  for (int i = 0; i < n; i++) {
    int x, y;
    cin >> x >> y;
    A[i] = make_tuple(x*x + y*y, -x, -y);
  }

  // sortiramo tacke
  sort(A.begin(), A.end());

  // ispisujemo tacke
  for (int i = 0; i < n; i++)
    cout << -get<1>(A[i]) << " " << -get<2>(A[i]) << endl;

  return 0;
}
```

Задатак: Највреднији предмети

За сваки предмет који је на продају дата је шифра и цена. Купац има на располагању одређени износ динара и жели да купи што скупље предмете. Редом узима предмете почев од најскупљег, док има новца. Ако нема новца за најскупљи, узима најскупљи за који има новца. Приказати шифре предмета које купац купује и, ако му је остало, преостали износ новца. Напомена: ова стратегија не гарантује да ће предмети које купи бити укупно највеће могуће вредности (нпр. ако има 5 динара и ако су цене предмета 4, 3 и 2 динара, он ће купити предмет само предмет од 4 динара, а могао би да купи предмете од 3 и 2 динара).

Улаз: У првој линији стандардног улаза налази се износ новца (реалан број) који има купац, у другој број предмета, N, а затим се, у сваке две линије стандардног улаза, уносе, редом, шифра (ниска карактера) па цена (реалан број) предмета, свака у посебном реду, за свих N предмета.

Излаз: У свакој линији стандарног излаза исписују се шифре и цене купљених предмета (раздвојене размаком), ако их има. У последњој линији приказује се преостали износ новца, ако постоји.

Пример1

Улаз	*Излаз*

`1250.75`
`5`
`predmet1`
`1010.30`
`predmet2`
`357.35`
`predmet3`
`725.45`
`predmet4`
`1125.5`
`predmet5`
`115.75`
	`predmet4 1125.5`
	`predmet5 115.75`
	`9.50`

Пример2

Улаз	*Излаз*

`10000`
`6`
`predmet1`
`3010`
`predmet2`
`3005`
`predmet3`
`5725`
`predmet4`
`1265`
`predmet5`
`2075`
`predmet6`
`385`
	`predmet3 5725.00`
	`predmet1 3010.00`
	`predmet4 1265.00`

Задатак: Сортирање линија

Написати програм који уређује линије учитане са стандардног улаза, лексикографски, по абецедном редоследу, не правећи разлику између великих и малих слова.

Улаз: Са стандардног улаза се учитавају линије све до краја улаза (при интерактивном уносу, крај улаза се може постићи тастерима `ctrl + Z` тј. `ctrl + D`).

Излаз: На стандардном излазу се приказује сортиране учитане линије.

Пример

Улаз	*Излаз*
`Pera`	`ananas`
`ananas`	`Mika`
`Mika`	`Pera`
`pertla`	`pertla`

Решење

Приметимо да је овај задатак веома сличан задаку Лексикографски минимум, једино што је у том задатку било довољно пронаћи само најмањи елемент (минимум), док се у овом задатку тражи сортирање свих елемената.

Учитавање линија до краја улаза вршићемо на сличан начин оном приказаном у задатку Читање до краја улаза. Пошто у задатку није речено колико ће линија бити унето, за њихово смештање користићемо динамички низ у који можемо да додајемо елементе на крај. У језику C++ то може бити вектор типа `vector<string>`. Сваку учитану линију ћемо у вектор додавати методом `push_back`.

Сортирање библиотечком функцијом

У језику C++ сортирање ћемо вршити библиотечком функцијом `sort`. Као аргумент ћемо наводити функцију поређења, која врши поређење линија игноришући разлику између великих и малих слова. Разне начине да се функцији сортирања проследи функција поређења видели смо у задатку Сортирање такмичара.

Сложеност библиотечке функције сортирања квазилинеарна тј. $O(n \log n)$, али у терминима броја поређења и размена елемената, а не елементарних операција. Пошто се у овом задатку ниске пореде лексикографски, сложеност најгорег случаја поређења (када су ниске једнаке) је $O(k)$, где је k највећа дужина две ниске. Стога је укупна сложеност овог програма $O(kn \log n)$. Ипак, ако су ниске "шаренолике", обично се већ на основу првих неколико слова одреди која је лексикографски већа, па овај најгори случај ретко кад наступа.

Функцију поређења ниски можемо дословно преузети из решења задатка Лексикографски минимум. Она може ручно имплементирати алгоритам лексикографског поређења.

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
```

```
#include <algorithm>
using namespace std;


int poredi_niske(const string& a, const string& b) {
  for (int i = 0; i < a.size() && i < b.size(); i++)
    if (tolower(a[i]) != tolower(b[i]))
      return tolower(a[i]) - tolower(b[i]);
  return a.size() - b.size();
}

bool manja_niska(const string& a, const string& b) {
  return poredi_niske(a, b) < 0;
}

int main() {
  // vektor u kojem pamtimo sve ucitane linije
  vector<string> linije;
  // ucitavamo liniju po liniju sve do kraja ulaza
  string linija;
  while (cin >> linija)
    // dodajemo liniju u vektor
    linije.push_back(linija);

  // sortiramo vektor
  sort(linije.begin(), linije.end(), manja_niska);

  // ispisujemo sortirane linije
  for (string linija: linije)
    cout << linija << endl;

  return 0;
}
```

Пошто се ниске пореде лексикографски, могуће да употребимо библиотечку функцију `lexicographical_compare`. Она прима два распона (ограничена паровима итератора) које лексикографски пореди, при чему се поређење појединачних елемената може вршити функцијом која се задаје као последњи аргумент. На том месту можемо проследити функцију која пореди карактере занемарујући разлику између малих и великих слова (тако што пре поређења оба карактера поретвори у мала слова, коришћењем библиотечке функције `tolower`).

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
#include <algorithm>
using namespace std;

bool poredi_karaktere(char a, char b) {
  return tolower(a) < tolower(b);
}

bool poredi_niske(const string& a, const string& b) {
  return lexicographical_compare(a.begin(), a.end(),
                                 b.begin(), b.end(),
                                 poredi_karaktere);
}

int main() {
```

```
  // vektor u kojem pamtimo sve ucitane linije
  vector<string> linije;
  // ucitavamo liniju po liniju sve do kraja ulaza
  string linija;
  while (cin >> linija)
    // dodajemo liniju u vektor
    linije.push_back(linija);

  // sortiramo vektor
  sort(linije.begin(), linije.end(), porediNiske);

  // ispisujemo sortirane linije
  for (string linija: linije)
    cout << linija << endl;

  return 0;
}
```

Уместо именованих, могуће је користити и анонимне функције поређења.

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
#include <algorithm>
using namespace std;

int main() {
  // vektor u kojem pamtimo sve ucitane linije
  vector<string> linije;
  // ucitavamo liniju po liniju sve do kraja ulaza
  string linija;
  while (cin >> linija)
    // dodajemo liniju u vektor
    linije.push_back(linija);

  // sortiramo vektor
  sort(linije.begin(), linije.end(),
       [](const string& a, const string& b) {
         return lexicographical_compare(a.begin(), a.end(),
                                        b.begin(), b.end(),
                                        [](char ca, char cb) {
                                          return tolower(ca) < tolower(cb);
                                        });
       });

  // ispisujemo sortirane linije
  for (string linija: linije)
    cout << linija << endl;

  return 0;
}
```

Задатак: Сортирање по просеку

Наставник је записао табелу са закључним оценама ученика. Напиши програм који сортира ученике по просечној оцени, нерастуће. Ако два ученика имају исту просечну оцену они треба да остану у редоследу у ком су били на почетку.

Улаз: Са стандардног улаза се учитава број ученика u ($5 \leq u \leq 50000$), затим број оцена o ($5 \leq o \leq 100$) и

након тога у u наредних линија оцене за сваког ученика (o оцена раздвојене размацима).

Излаз: На станардном излазу исписати соритране оцене (у истом формату у којем су и унете).

Пример

Улаз	*Излаз*
6 5	5 5 5 4 5
3 5 5 4 2	5 4 5 5 5
5 4 2 2 5	3 5 5 4 2
5 5 5 4 5	5 4 2 2 5
3 3 3 2 1	4 2 5 1 3
4 2 5 1 3	3 3 3 2 1
5 4 5 5 5	

Решење

Прво питање је избор структуре за складиштење свих података. Један начин је да се употреби обична матрица (статички алоцирана или вектор вектора).

Сортирање библиотечком функцијом

Сортирање можемо реализовати библиотечком функцијом, међутим, морамо обратити пажњу на то да се у тексту задатка очекује стабилно сортирање (задржавање оригиналног редоследа у случају када су просеци једнаки). У језику C++ то се може постићи функцијом `stable_sort` која се користи на исти начин као и функција `sort`. Као аргумент ћемо навести и функцију поређења (слично као у задатку Сортирање такмичара).

Функција поређења треба да израчуна просек оцена ученика који се пореде. Пошто сви имају једнак број оцена, онда је просек оцена првог ученика већи од просека оцена другог ако и само ако је збир његових оцена већи од збира оцена другог. Дакле, надаље можемо разматрати поређење на основу збира оцена. Најједноставнији начин имплементације је да се у функцији поређења сваки пут изнова рачуна збир оцена ученика који се пореде.

Библиотечка функција сортирања функцију поређења позива $O(n \log n)$ пута. Пошто се приликом сваког поређења рачуна збир оцена ученика, сложеност поређења је $O(m)$, где је m укупан број оцена ученика. Сложеност сортирања је, дакле, $O(mn \log n)$. Међутим, треба поменути време потребно за учитавање и испис података. Иако је сложеност сваке од те два фазе $O(mn)$, у пракси оно може апсолутно доминирати радом целог програма (јер је приступ периферијским уређајима често много спорије него приступ подацима у главној меморији).

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <numeric>

using namespace std;

void ucitaj(vector<vector<int>>& ocene) {
  int brojUcenika, brojOcena;
  cin >> brojUcenika >> brojOcena;
  ocene.resize(brojUcenika);
  for (int u = 0; u < brojUcenika; u++) {
    ocene[u].resize(brojOcena);
    for (int o = 0; o < brojOcena; o++)
      cin >> ocene[u][o];
  }
}

void ispisi(const vector<vector<int>>& ocene) {
  for (int u = 0; u < ocene.size(); u++) {
    for (int o = 0; o < ocene[u].size(); o++)
      cout << ocene[u][o] << " ";
    cout << endl;
```

```
  }
}

void sortiraj(vector<vector<int>>& ocene) {
  stable_sort(ocene.begin(), ocene.end(),
              [](const vector<int>& u1, const vector<int>& u2) {
                return accumulate(u1.begin(), u1.end(), 0) >
                       accumulate(u2.begin(), u2.end(), 0);
              });
}


int main() {
  vector<vector<int>> ocene;
  ucitaj(ocene);
  sortiraj(ocene);
  ispisi(ocene);
}
```

Оптимизација функције поређења

Рачунање збира оцена изнова може бити неефикасно (напоменимо да у овом конкретном задатку највише времена потроши учитавање и исписивање података, тако да се ова разлика у ефикасности не може приметити). Ако се оцене чувају у матрици, онда је тој матрици могуће додати посебну, последњу колону у којој би биле уписани збирови оцена сваког ученика и врсте те матрице сортирати на основу њихових последњих елемената. Збирове је могуће сачувати и у посебном низу и приликом поређења је могуће консултовати чланове тог низа (међутим, то може бити компликовано уколико се жели употребити библиотечка функција).

Израчунавање збирова оцена свих ученика се врши у сложености $O(mn)$, где је m број оцена, а n број ученика. Након тога, поређење се врши у константној сложености, па је сложеност сортирања $O(n \log n)$. Укупна сложеност је, дакле, $O(mn + n \log n)$. Наравно, у пракси и у овој имплементацији доминира учитавање и испис свих података.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <numeric>

using namespace std;

void ucitaj(vector<vector<int>>& ocene) {
  int brojUcenika, brojOcena;
  cin >> brojUcenika >> brojOcena;
  ocene.resize(brojUcenika);
  for (int u = 0; u < brojUcenika; u++) {
    ocene[u].resize(brojOcena);
    for (int o = 0; o < brojOcena; o++)
      cin >> ocene[u][o];
  }
}

void ispisi(const vector<vector<int>>& ocene) {
  for (int u = 0; u < ocene.size(); u++) {
    for (int o = 0; o < ocene[u].size(); o++)
      cout << ocene[u][o] << " ";
    cout << endl;
  }
}

void sortiraj(vector<vector<int>>& ocene) {
```

```
  // dodajemo zbir ocena na kraj ocena svakog ucenika
  for_each(ocene.begin(), ocene.end(), [](vector<int>& ucenik) {
      ucenik.push_back(accumulate(ucenik.begin(), ucenik.end(), 0));
    });
  // sortiramo ucenike na osnovu poslednje vrednost - to je upravo dodati zbir
  stable_sort(ocene.begin(), ocene.end(),
              [](const vector<int>& u1, const vector<int>& u2) {
                  return u1.back() > u2.back();
              });
  // uklanjamo zbirove sa kraja
  for_each(ocene.begin(), ocene.end(), [](vector<int>& ucenik) {
      ucenik.pop_back();
    });
}

int main() {
  ios_base::sync_with_stdio(false);
  vector<vector<int>> ocene;
  // ucitavamo ocene ucenika
  ucitaj(ocene);
  // sortiramo ucenike na osnovu ocena
  sortiraj(ocene);
  // ispisujemo sortirane ocene
  ispisi(ocene);
}
```

Сортирање низа структура

Једна могућност је да оцене сваког ученика спакујемо у структуру која садржи низ оцена (и евентуално још неке податке). Ако се подаци о оценама ученика памте у склопу структуре, тада је у тој структури могуће чувати и збир свих оцена и сортирање вршити на основу њега.

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

struct Ucenik {
  vector<int> ocene;
  int zbir;
};

void ucitaj(vector<Ucenik>& ucenici) {
  int brojUcenika, brojOcena;
  cin >> brojUcenika >> brojOcena;
  ucenici.resize(brojUcenika);
  for (int u = 0; u < brojUcenika; u++) {
    ucenici[u].ocene.resize(brojOcena);
    ucenici[u].zbir = 0;
    for (int o = 0; o < brojOcena; o++) {
      cin >> ucenici[u].ocene[o];
      ucenici[u].zbir += ucenici[u].ocene[o];
    }
  }
}

void ispisi(const vector<Ucenik>& ucenici) {
  for (const Ucenik& u : ucenici) {
    for (int ocena : u.ocene)
```

```
      cout << ocena << " ";
    cout << endl;
  }
}

void sortiraj(vector<Ucenik>& ucenici) {
  // sortiramo ucenike na osnovu zbira ocena
  stable_sort(ucenici.begin(), ucenici.end(),
              [](const Ucenik& u1, const Ucenik& u2) {
                  return u1.zbir > u2.zbir;
              });
}

int main() {
  vector<Ucenik> ucenici;
  // ucitavamo ocene ucenika
  ucitaj(ucenici);
  // sortiramo ucenike na osnovu ocena
  sortiraj(ucenici);
  // ispisujemo sortirane ocene
  ispisi(ucenici);
}
```

Задатак: Ранг сваког елемента

Након такмичења из информатике позната је листа ученика са поенима (сваки ученик има различит број поена), међутим, није познат пласман (редни број сваког ученика). Напиши програм који за сваког ученика одређује пласман и исписује извештај о пласману, при чему су ученици приказани у истом редоследу као и у полазној листи са поенима.

Улаз: Са стандардног улаза уноси се број n, а затим и подаци о n ученика (за сваког ученика су у посебном реду дати име и број поена раздвојени размаком).

Излаз: На стандардни излаз за сваког ученика у посебном реду треба исписати име и пласман раздвојене размаком.

Пример

Улаз	*Излаз*
5	Pera 3
Pera 85	Ana 1
Ana 93	Jelena 2
Jelena 90	Mika 5
Mika 72	Lidija 4
Lidija 75	

2.8.3 Сортирање разврставањем

Сваки алгоритам сортирања који сортирање низа врши тако што размењује елементе низа који се сортира (каже се "сортира низ у месту, размена елемената") је бар квази-линеарне сложености и захтева у најгорем случају улаза бар $O(n \log n)$ операција поређења. Ипак, у неким специфичним ситуацијама (када елементи низа задовољавају неке унапред задате услове) могуће је дизајнирати алгоритме сортирања који су линеарне сложености $O(n)$. Ти алгоритми се углавном заснивају на пребројавању елемената у низу и често ангажују додатну меморију.

Основни алгоритам соритања линеарне сложености је *сортирање пребројавањем* (енгл. *Counting sort*) и он се примењује када знамо да су сви елементи у низу из неког ограниченог опсега вредности и да се по правилу јављају више пута у низу. Основна идеја је да се просто помоћу низа бројача (асоцијативног низа, хеш-таблице) преброји колико се пута у низу јавља сваки од могућих елемената и да се затим низ реконструише на основу те информације (уместо размена, у низ се директно уписује нови садржај).

Други алгоритам је *сортирање разврставањем* и *сортирање вишеструким разврставањем* (енгл. *Radix sort* или *Bucket sort*). Идеја сортирања разврставањем заснована је на томе да се елементи низа групишу на основу

неког критеријума (који је у складу са критеријумом сортирања), а да се затим елементи сваке групе сортирају засебно. Чувени пример је *Radix sort* код ког се, на пример, низ троцифрених бројева сортира тако што се у првој фази елементи сортирају *Counting sort* алгоритмом само по цифри јединица, затим истим алгоритмом само по цифри десетица и на крају само по цифри стотина. При томе се води рачуна да се у накнадним фазама редослед еквивалентних елемента не промени. На пример, приликом сортирања по цифри десетица, елементи који имају исту цифру десетица треба да остану у редоследу у ком су били (такво сортирање се назива стабилним). Читаоцу остављамо за вежбу да размисли зашто је овакав поступак коректан, односно зашто се његовом применом добија сортиран низ.

Ово се, наравно, природно уопштава и на вишецифрене бројеве. Такође, основа система записивања која се користи не мора да буде 10 (може се, на пример, користити систем основе 16 или основе 256).

Задатак: Сортирање пребројавањем

Низ садржи бројеве из интервала од 0 до $m-1$. Напиши програм који сортира тај низ.

Улаз: Са стандардног улаза уноси се број m ($1 \leq m \leq 1000$), број n ($1 \leq n \leq 10^5$) и затим n бројева из интервала од 0 до $m-1$.

Излаз: На стандардни излаз исписати сортиране елементе низа.

Пример

Улаз	*Излаз*
4	1
6	1
3	2
1	2
2	3
3	3
2	
1	

Решење

Сортирање пребројавањем

Сортирање пребројавањем има смисла када је распон могућих елемената у низу мали (што је овде случај јер су елементи увек из интервала $[0, 1000]$) и када се очекује да се елементи у низу понављају. Основна идеја сортирања пребројавањем је да се за сваки могући елемент преброји колико пута се јавља у низу. То можемо урадити на сличан начин као у задатку Фреквенција знака или Фреквенције речи. Када се зна број појављивања сваког елемента, тада се сортиран низ може реконструисати тако што се сваки елемент у резултујући низ дода онолико пута колико се јавио у полазном низу. Реконструкцију дакле, вршимо помоћу угнежђених петљи где спољна петља пролази кроз све елементе из распона (у нашем случају кроз вредности од 0 до m), док се унутрашња извршава онолико пута колико је број појављивања текуће вредности бројача спољне петље. У телу унутрашње петље се бројач спољашње петље додаје на текућу позицију у резултујућем низу (која се пре петљи иницијализује на нулу и приликом додавања сваког елемента увећава за 1).

Сложеност фазе пребројавања је $O(n)$, а сложеност реконструкције је $O(m)$, па је укупна сложеност једнака $O(n+m)$. Ако m није за ред величине веће у односу на n, овај алгоритам је практично линеарне сложености, што је мало боље него класични алгоритми сортирања. Пошто се користи један низ димензије n и један низ димензије m, меморијска сложеност је $O(n+m)$.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo broj mogucih razlicitih elemenata niza
  int m;
  cin >> m;
```

```
  // ucitavamo elemente niza
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // brojimo pojavljivanja svakog elementa niza
  vector<int> frekvencije(m, 0);
  for (int i = 0; i < n; i++)
    frekvencije[a[i]]++;

  // rekonstruisemo niz iz pocetka
  int k = 0;
  for (int j = 0; j < m; j++)
    for (int i = 0; i < frekvencije[j]; i++)
      a[k++] = j;

  // ispisujemo elemente niza
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;

  return 0;
}
```

Уштеда меморије

Приметимо да је у програму заправо било могуће одржавати само низ бројача док ни улазни ни резултујући низ није било потребно памтити (елементи се могу бројати приликом учитавања и уместо додавања у резултат одмах исписивати). Ако је распон елемената доста мањи од дужине низа, ово значи да се сортирањем пребројавањем доста штеди меморија.

Меморијска сложеност овог приступа је $O(m)$, док временска сложеност остаје $O(m+n)$.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo broj mogucih razlicitih elemenata niza
  int m;
  cin >> m;

  // ucitavamo elemente niza
  int n;
  cin >> n;

  // brojimo elemente ne smestajuci ih u niz
  vector<int> frekvencije(m, 0);
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    frekvencije[x]++;
  }

  // rekonstruisemo elemente sortiranog niza
```

```
  for (int j = 0; j < m; j++)
    for (int i = 0; i < frekvencije[j]; i++)
      cout << j << endl;

  return 0;
}
```

Класично сортирање

Наравно, задатак је могуће решити и било којим класичним алгоритмом сортирања који смо приказали у задатку Сортирање бројева.

Сложеност овог приступа је квази-линеарна тј. $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo broj mogucih razlicitih elemenata niza
  int m;
  cin >> m;

  // ucitavamo elemente niza
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // sortiramo elemente niza
  sort(a.begin(), a.end());

  // ispisujemo elemente niza
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;

  return 0;
}
```

Задатак: Разврставање по општинама

Државна комисија је направила списак свих такмичара у држави. Потребно је да се свакој општини дистрибуира списак такмичара са територије те општине, али тако да редослед остане исти какав је на полазном списку државне комисије.

Улаз: Са стандардног улаза се уноси списак такмичара, све до краја улаза. За сваког такмичара се уноси назив општине и шифра такмичара, раздвојене једним табулатором (карактером `Tab`). У тексту су коришћени само ASCII карактери. Напомена: приликом интерактивног тестирања крај улаза се може унети помоћу `ctrl + z` тј. `ctrl + d`. Број такмичара је највише 10^5, а број општина највише 10^3, док су дужина наизва такмичара и општине највише 15 карактера.

Излаз: На стандардни излаз исписати спискове за све општине, сваки у посебном реду. Општине су уређене лексикографски, растуће. Сваки списак почиње називом општине који је праћен знаком `:`. Након тога се наводе шифре свих такмичара раздвојене запетама (знацима `,`). Иза сваког знака интерпункције (знака `:` и знака `,`) наводи се по један размак.

Пример

Улаз

```
vozdovac    programmer
svilajnac   teamX
vozdovac    astro
alibunar    mika
svilajnac   pera.peric
medijana    luke skywalker
```

Излаз

```
alibunar: mika
medijana: luke skywalker
svilajnac: teamX, pera.peric
vozdovac: programmer, astro
```

Решење

Сортирање разврставањем

Најефикаснији начин да извршимо разврставање елемената по општинама је линеарне сложености, али подразумева да се елементи из полазног премештају у помоћни низ. Разврставање вршимо у неколико пролаза. У првом одређујемо број такмичара у свакој општини. У другом одређујемо позиције у низу на које ћемо смештати такмичаре из сваке општине, док у трећем вршимо пребацивање из полазног у резултујући низ (ажурирајући при том позиције такмичара за сваку општину).

Дакле, први пролаз је сличан као код сортирања пребројавањем које смо срели у задатку Сортирање пребројавањем.

За пример дат у тексту задатка, у првом пролазу одређујемо наредне бројеве такмичара за сваку општину.

```
alibunar: 2
medijana: 1
svilajnac: 2
vozdovac: 2
```

Позицију првог такмичара из сваке општине у резултујућем низу можемо одредити као укупан број такмичара у свим општинама које јој претходе (последњу позицију добијамо ако урачунамо и број такмичара у тој општини). Те позиције можемо одредити инкрементално (као, на пример, у задатку Префикс највећег збира).

На тај начин добијамо следеће позиције

```
alibunar: 0
medijana: 2
svilajnac: 3
vozdovac: 5
```

Приликом преписивања пролазимо редом кроз елементе полазног низа и смештамо их у резултујући низ на позицију придружену тој општини, уједно повећавајући ту позицију за 1, како би наредни елемент дошао на своје место.

Тако се `programer` уписује на место број 5, а позиција наредног такмичара са општине `vozdovac` се повећава на 6, затим се `teamX` уписује на место број 3, а позиција наредног такмичара са општине `svilajnac` се повћева на 4, затим се `astro` upisuje na mesto broj 6, а позиција за општину `vozdovac` се повећава на 7 итд.

Из приказаног примера је јасно да је свакој општини је потребно придружити број такмичара са те општине, а након тога и позицију у резултујућем низу на коју ће се приликом преписивања из полазног у резултујући низ придружити наредни такмичар из те општине. Пошто списак општина није унапред фиксиран, за та придруживања морамо користимо пресликавање (мапу, речник). Овакве структуре података смо већ користили у задатку Фреквенције речи. У језику C++ можемо користити мапу тј. `map<string, int>`.

Сложеност алгоритма зависи од броја такмичара и броја општина. Ако претпоставимо да је број такмичара n много већи од броја општина (што је прилично реална претпоставка), можемо закључити да ће временом доминирати два пролаза кроз оригинални низ и под претпоставком да ажурирање података придружених општинама тумачимо као да су операције константне сложености, укупна сложеност ће бити $O(n)$ (услед малог броја општина, чак и ако се користе мапе тј. речници који умеђу и приступају елементима у логаритамском времену, фактор $\log m$ је толико мали да га можемо сматрати практично константним). Не треба занемарити ни време поређења ниски, међутим, иако је оно у најгорем случају линеарно у односу на дужину ниски које се пореде, можемо очекивати да су имена веома кратка и "шаренолика" и да се поредак може одредити већ након поређења првих неколико карактера (па поређење грубо можемо проценити операцијом константне сложености).

Приметимо да је претходни алгоритам изводио *стабилно сортирање* тј. да је редослед свих такмичара са исте општине задржан.

```
#include <iostream>
#include <map>
#include <vector>
#include <string>

using namespace std;

// za svakog takmicara je poznata opstina sa koje dolazi i sifra
struct Takmicar {
  string opstina;
  string sifra;
};

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavanje vektora takmicara
  vector<Takmicar> takmicari;
  string s;
  while (getline(cin, s)) {
    int p = s.find('\t');
    string opstina = s.substr(0, p);
    string sifra = s.substr(p + 1);
    takmicari.push_back({opstina, sifra});
  }

  // izracunavamo broj takmicara iz svake opstine
  // za dati naziv opstine brojTakmicara odredjuje broj takmicara
  map<string, int> brojTakmicara;
  // prolazimo kroz spisak svih takmicara
  for (auto takmicar : takmicari)
    // uvecavamo broj takmicara na opstini trenutnog takmicara
    brojTakmicara[takmicar.opstina]++;

  // izracunavamo poziciju u sortiranom nizu na kojoj
  // pocinju takmicari sa date opstine
  map<string, int> pozicije;
  // ukupan broj takmicara u do sada obradjenim opstinama
  int prethodnoTakmicara = 0;
  // prolazimo kroz sve opstine u sortiranom redosledu
  // (abecedno, leksikografski)
  for (auto it : brojTakmicara) {
    // tekuca opstina pocinje na poziciji odredjenoj
    // brojem takmicara u prethodnim opstinama
    pozicije[it.first] = prethodnoTakmicara;
    // uvecavamo broj takmicara u prethodnim opstinama za broj
    // takmicara u trenutnoj opstini, pripremajuci se za novu
    // iteraciju
    prethodnoTakmicara += it.second;
  }

  // konacan sortirani niz takmicara
  vector<Takmicar> sortirano(takmicari.size());
  // prolazimo kroz sve takmicare iz polaznog niza
  for (auto takmicar : takmicari) {
    // postavljamo takmicara na tekuce mesto u njegovoj opstini
```

```
    sortirano[pozicije[takmicar.opstina]] = takmicar;
    // uvecavamo slobodnu poziciju u toj opstini
    pozicije[takmicar.opstina]++;
  }

  // ispisujemo konacni spisak u trazenom formatu
  cout << sortirano[0].opstina << ": " << sortirano[0].sifra;
  for (int i = 1; i < sortirano.size(); i++)
    if (sortirano[i].opstina != sortirano[i-1].opstina)
      cout << endl << sortirano[i].opstina << ": " << sortirano[i].sifra;
    else
      cout << ", " << sortirano[i].sifra;
  cout << endl;

  return 0;
}
```

Библиотечке функције сортирања

У решењу можемо применити библиотечке функције за стабилно сортирање (описане у задатку Разврставање по првом слову). Ипак, уместо тога демонстрираћемо да се стабилно сортирање може остварити и обичном библиотечком функцијом сортирања, коју смо описали у задатку Сортирање такмичара). У језику C++ то је функција `sort`. Та функција не мора обавезно бити стабилна, међутим стабилност можемо наметнути кроз функцију поређења која одређује редослед сортирања. У структуру којом се репрезентују такмичари, поред општине и шифре такмичара уписаћемо и редни број такмичра на оригиналном списку. Функција поређења онда прво пореди општину, а ако су два такмичара који се пореде из исте општине, редослед одређује на основу редних бројева са оригиналног списка (овакво хијерархијско, тј. лексикографско поређење смо увели у задатку Пунолетство).

Сложеност сортирања је $O(n \log n)$. При том треба узети у обзир и време поређења две ниске, међутим, пошто су оне веома кратке и "шаренолике", реално је очекивати да се поредак може одредити већ након поређења првих неколико карактера (па поређење грубо можемо проценити операцијом константне сложености).

```
#include <iostream>
#include <map>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// za svakog takmicara je poznata opstina sa koje dolazi i sifra
// pamtimo jos i redni broj u polaznom nizu
struct Takmicar {
  string opstina;
  string sifra;
  size_t redniBroj;
};

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavanje vektora takmicara
  vector<Takmicar> takmicari;
  string s;
  while (getline(cin, s)) {
    int p = s.find('\t');
    string opstina = s.substr(0, p);
    string sifra = s.substr(p + 1);
    takmicari.push_back({opstina, sifra, takmicari.size()});
```

```
  }

  // sortiranje takmicara
  sort(takmicari.begin(), takmicari.end(),
       [](const Takmicar& t1, const Takmicar& t2) {
         // takmicari se sortiraju najpre na osnovu opstine,
         // a u okviru iste opstine na osnovu rednog broja u originalnom nizu
         return t1.opstina < t2.opstina ||
                (t1.opstina == t2.opstina && t1.redniBroj < t2.redniBroj);
       });

  // ispisujemo konacni spisak u trazenom formatu
  cout << takmicari[0].opstina << ": " << takmicari[0].sifra;
  for (int i = 1; i < takmicari.size(); i++)
    if (takmicari[i].opstina != takmicari[i-1].opstina)
      cout << endl << takmicari[i].opstina << ": " << takmicari[i].sifra;
    else
      cout << ", " << takmicari[i].sifra;
  cout << endl;

  return 0;
}
```

Задатак: Разврставање по првом слову

Ђаци су дошли на систематски преглед. Пошто их је јако пуно, прегледаће их више лекара, а ученици ће се делити на основу њихових иницијала (почетних слова имена и презимена). Информациони систем је веома стар, нажалост, користи само латиничка слова енглеске абецеде. Сестра је добила задатак да поређа прикупљене књижице тако да прво иде гомила књижица за децу чији су иницијали `aa`, затим `ab`, затим `ac` и тако даље, затим `ba`, `bb`, `bc` и тако даље, па све до `zy` и `zz`. Унутар сваке групе (сваког пара иницијала) редослед књижица мора да остане исти као и у почетном низу.

Улаз: Са стандардног улаза учитавају се имена и презимена ученика - по једно име и презиме у свакој линији, раздвојени тачно једним размаком, све до краја улаза. Напомена: приликом интерактивног тестирања крај улаза се може унети помоћу `ctrl + z` тј. `ctrl + d`.

Излаз: На стандардни излаз исписати сва учитана имена и презимена у траженом редоследу.

Пример

Улаз	*Излаз*
`branko zdravkovic`	`aleksandar aleksic`
`aleksandar aleksic`	`aleksije avakumovic`
`mirko svetic`	`andrea bircanin`
`aleksije avakumovic`	`branko zdravkovic`
`cile petrovic`	`cile petrovic`
`marko savic`	`mirko svetic`
`andrea bircanin`	`marko savic`

Задатак: Сортирање бројева вишеструким разврставањем (RadixSort)

Дат је низ природних бројева. Имплементирај програм који их сортира алгоритмом вишеструког разврставања (RadixSort).

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим n природних бројева између 1 и 999999999 (сваки у посебном реду).

Излаз: На стандардни излаз испиши те бројеве у неопадајућем редоследу.

Пример

Улаз	*Излаз*
9	43
342	342
43	432
5432	546
432	636
34222	3423
3423	5432
546	6363
6363	34222
636	

Решење

У задатку Разврставање по првом слову видели смо да се разврставање може применити више пута (под претпоставком да се имплементира на стабилан начин) да би се низ сортирао у лексикографском редоследу кључева представљених одређеним n-торкама, задржавајући при том оригинални редослед елемената чији су кључеви једнаки.

Ако замислимо да су природни бројеви допуњени водећим нулама тако да су сви бројеви који се сортирају са истим бројем цифара, сортирање природних бројева се своди на сортирање торки у лексикографском редоследу и може се извршити вишеструким разврставањем, тако што се сортирање врши на основу сваке декадне позиције, кренувши од цифре јединица.

Прикажимо како се сортирање разврставањем може применити на следећи низ бројева.

```
37 140 7 10 99 102 17 25 1 48 14 3 18
```

Прво вршимо сортирање на основу цифре јединица

```
140 10 1 102 3 14 25 37 7 17 48 18 99
```

Након тога, на основу цифре десетица.

```
1 102 3 7 10 14 17 18 25 37 140 48 99
```

На крају, сортирање вршимо на основу цифре стотина.

```
1 3 7 10 14 17 18 25 37 48 99 102 140
```

Свако појединачно разврставање вршимо на уобичајени начин (као, на пример, у задатку Разврставање по општинама). У првом пролазу пребројимо елементе у свакој категорији (за сваку цифру од 0 до 9 пребројимо колико има елемената низа који на текућој декадној позицији садрже баш ту цифру). Након тога израчунамо парцијалне збирове тако добијеног низа и они нам за сваку категорију дају позицију у новом низу на коју треба поставити елемент из те категорије. На крају копирамо елементе из полазног у нови низ на одговарајуће позиције (ажурирајући те позиције након сваког уписа).

Свако појединачно разврставање врши се у линеарној сложености у односу на број елемената низа. Број позивања функције разврставања зависи од тога колико цифара имају бројеви који се сортирају (тј. колико цифара има највећи од њих). Пошто број цифара логаритамски зависи од величине броја, ако се сортира низ од n елемената у ком је највећи елемент m, тада се сортирање врши у сложености $O(n \log m)$. Поред низа користи се помоћни низ дужине n и низ бројача који је практично константне сложености (бројача има и колико и цифара). Стога је меморијска сложеност $O(n)$.

Уместо цифара у основи 10, могу се посматрати и "цифре" у другим бројевним основама. На пример, можемо посматрати групе од по 8-битова у бинарном запису броја и сортирати на основу "цифара" у основи 256 (тиме се број разврставања смањује на само 4 за 32-битне бројеве). Цена ангажовања мало већег броја бројача (256 уместо 10) је практично занемарива.

```
#include <iostream>
#include <vector>
#include <limits>

using namespace std;
```

```
void sortiranjeRazvrstavanjem(vector<int>& a, int s) {
  // brojimo pojavljivanje svake cifre uz koeficijent s (stepen desetke)
  int frekvencija[10] = {0};
  for (int i = 0; i < a.size(); i++)
    frekvencija[(a[i] / s) % 10]++;
  // pozicije u rezultujucem nizu ispred kojih se zavrsavaju grupe
  // elemenata sa odgovarajucim ciframa uz koeficijent s
  for (int i = 1; i < 10; i++)
    frekvencija[i] += frekvencija[i-1];
  // prepisujemo elemente u pomocni niz
  vector<int> pom(a.size());
  for (int i = a.size() - 1; i >= 0; i--)
    pom[--frekvencija[(a[i] / s) % 10]] = a[i];
  // vracamo elemente nazad u glavni niz
  a = pom;
}

void sortiranjeVisestrukimRazvrstavanjem(vector<int>& a) {
  // odredjujemo maksimum niza a
  int max = numeric_limits<int>::min();
  for (int x : a)
    if (x > max)
      max = x;

  // sortiramo na osnovu svake cifre krenuvsi od cifara jedinica
  for (int s = 1; max / s > 0; s *= 10)
    sortiranjeRazvrstavanjem(a, s);
}

int main() {
  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  sortiranjeVisestrukimRazvrstavanjem(a);

  // ispisujemo rezultat
  for (int x : a)
    cout << x << endl;

  return 0;
}
```

2.8.4 Обрада дупликата (поновљених вредности у низу)

У неким задацима је потребно на неки начин обрадити све поновљене вредности у низу (дупликате). Ефикасна решења се обично добијају након што се низ претпроцесира коришћењем сортирања. Након сортирања низа сви поновљени елементи се налазе један иза другога, што значајно онда олакшава њихову обраду (за сваки елемент је веома једноставно проверити колико пута се јавио у низу, па је самим тим једноставно проверити и да ли је дупликат, уклонити дупликате и слично). Осим сортирањем, обрада дупликата се може вршити и помоћу библиотечких колекција (скупова, мултискупова и мапа тј. речника), о чему ће више речи бити касније.

Задатак: Дупликати

Претпоставимо да су интернет адресе представљене природним бројевима (IP адресе се, на пример, чувају у облику неозначених 32-битних бројева). Претраживач чува списак свих адреса које је корисник посетио током неког претходног периода. Корисник је многе адресе посећивао и више пута. Напиши програм који одређује број различитих адреса које је корисник посетио.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$), а затим и n природних бројева (мањих од 2^32), сваки у посебном реду.

Излаз: На стандардни излаз исписати број различитих адреса које је корисник посетио.

Пример

Улаз	*Излаз*
8 123456789 234567890 345678901 234567890 456789012 234567890 456789012 234567890	4

Решење

Претрага

Наиван начин да се детектују дупликати се може засновати на алгоритму линеарне претраге. Од свих учитаних елемената низа бројаћемо само оне који се први пут појављују (када се неки елемент појави други, трећи итд. пут, нећемо га бројати). За сваки елемент низа на позицији i (од нула до $n - 1$) провераваћемо да ли се тај елемент јавља на некој позицији од 0 до $i - 1$. То можемо урадити алгоритмом линеарне претраге. Разне начине имплементације линеарне претраге описали смо у задатку Негативан број. Линеарна претрага се може имплементирати и библиотечком функцијом.

Ово решење је неефикасно и сложеност му је квадратна у односу на димензију низа (сложеност му је $O(n^2)$, где је n број елемената низа).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo niz
  int n;
  cin >> n;
  vector<unsigned> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int brojRazlicitih = 0;
  // za svaki element niza a
  for (int i = 0; i < n; i++) {
    // proveravamo da li se a[i] javlja pre pozicije i
    bool sadrzi = false;
    for (int j = 0; j < i; j++)
      if (a[i] == a[j]) {
        sadrzi = true;
        break;
      }
```

```
    // ako se ne pojavljuje, uracunavamo ga
    if (!sadrzi)
      brojRazlicitih++;
  }
  cout << brojRazlicitih << endl;
  return 0;
}
```

Сортирање

Један од најчешћих начина уклањања дупликата из низа је заснован на сортирању, јер се након сортирања дупликати нађу један до другог. Сортирање можемо најбоље урадити позивом библиотечке функције. Разни начини на које је могуће сортирати низ бројева су приказани у задатку Сортирање бројева. Након сортирања пролазимо редом кроз низ и бројимо први елемент, а затим и све елементе који су различити од њима претходног (то су прва појављивања елемената у сортираном низу).

Сложеношћу овог приступа доминира сложеност поступка сортирања. Пролаз након сортирања је линеарне сложености, а сортирање се може остварити у сложености $O(n \log n)$, где је n број елемената низа.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo niz
  int n;
  cin >> n;
  vector<unsigned> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  // sortiramo niz
  sort(a.begin(), a.end());
  // brojimo prvi element i sve elemente koje su razliciti od
  // svojih prethodnika
  int brojRazlicitih = 1;
  for (int i = 1; i < a.size(); i++)
    if (a[i] != a[i-1])
      brojRazlicitih++;
  cout << brojRazlicitih << endl;

  return 0;
}
```

Библиотечка функција за избацивање дупликата

Избацивање дупликата у сортираном низу може се вршити и библиотечким функцијама. Функција `unique` у језику C++ прима два итератора који ограничавају сегмент сортираног низа тј. вектора (обично `a` и `next(a, n)` тј. `a+n` или `a.begin()` и `a.end()`) и организује тај низ тако да на његов почетак смешта елементе који се не понављају и враћа итератор на позицију иза краја тог дела. На пример, ако је низ $1, 1, 2, 2, 5, 5$, након позива функције `unique` он ће бити реорганизован тако што ће му садржај бити $1, 2, 5, x, x, x$, а итератор ће указивати на позицију три тј. непосредно иза елемента 5. Саржај на позицијама обележеним са x није релевантан. Ако се жели избацивање дупликата, реп низа се може одстранити тако што се позове функција `erase` којој се наведу два итератора - први је онај који је вратила функција `unique` а други је итератор на крај низа (који се обично добија са `next(a, n)`, `a + n` или `a.end()`). Ако је потребно израчунати само број јединствених елемената он се може добити тако што се израчуна растојање између итератора на почетак низа и итератора којег врати функција `unique` (на пример, `distance(a, unique(a, next(a, n)))`).

Библиотечке функције за уклањање дупликата из сортиране колекције сложености су $O(n)$, па укупном сложеношћу доминира сортирање сложености $O(n \log n)$.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo niz
  int n;
  cin >> n;
  vector<unsigned> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  // sortiramo niz
  sort(a.begin(), a.end());
  // uklanjamo duplikate
  a.erase(unique(a.begin(), a.end()), a.end());
  // ispisujemo rezultat
  cout << a.size() << endl;

  // dovoljno je i:
  // cout << distance(a.begin(), unique(a.begin(), a.end())) << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Највећи поновљени елемент

Напиши програм који у низу бројева одређује највећи број који се појављује бар два пута у низу или констатује да такав број не постоји.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 50000$), а затим у наредних n редова n целих бројева између 1 и 50000.

Излаз: На стандардни излаз исписати тражени број или текст `nema`, ако су сви елементи низа различити.

Пример 1

Улаз	*Излаз*
6	3
3	
8	
2	
2	
3	
5	

Пример 2

Улаз	*Излаз*
3	nema
1	
2	
3	

Пример 3

Улаз	*Излаз*
6	3
3	
3	
2	
2	
1	
1	

Решење

Груба сила

Решење грубом силом подразумева да се за сваки елемент провери да ли се понавља бар два пута у низу (нпр. тако што се преброје његова појављивања или тако што се изврши претрага за текућим елементом на свим позицијама осим на текућој) и да се пронађе максимум елемената који се појављују више пута.

С обзиром на то да се анализира сваки од n елемената и да претрага (или пребројавање појављивања) у најгорем случају захтева обилазак скоро целог низа, сложеност овог алгоритма је $O(n^2)$, па је он недовољно ефикасан.

```cpp
#include <iostream>
#include <vector>
```

```
using namespace std;

int main() {
  // ucitavamo elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // najveci ponovljeni element
  int max = -1;

  // analiziramo jedan po jedan element niza
  for (int i = 0; i < n; i++) {
    // proveravamo da li se tekuci element ponavlja u nizu
    bool ponavljaSe = false;
    for (int j = 0; j < n; j++)
      if (i != j && a[i] == a[j]) {
        ponavljaSe = true;
        break;
      }
    // azuriramo maksimum ako je potrebno
    if (ponavljaSe && a[i] > max)
      max = a[i];
  }

  // prijavljujemo rezultat
  if (max != -1)
    cout << max << endl;
  else
    cout << "nema" << endl;

  return 0;
}
```

Сортирање

Алгоритам боље сложености се може добити ако се елементи низа претходно сортирају. Након тога могуће је елементе обрађивати са краја низа и за сваки елемент проверавати да ли је једнак претходном (ако се елемент понавља, након сортирања сва његова појављивања постају узастопна унутар низа). Пошто елемент обрађујемо у опадајућем редоследу, први елемент који се понавља уједно је и највећи такав елемент.

Сложеношћу доминира сортирање, које, ако се ефикасно изврши сложености $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  // ucitavamo element u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // sortiramo ih
```

```
  sort(begin(a), end(a));

  // obilazimo elemente u opadajucem redosledu
  int k;
  for (k = n-1; k > 0; k--)
    // element je ponovoljen akko je jednak prethodnom
    if (a[k] == a[k-1])  {
      // prvi ponovljeni je ujedno i najveci takav
      cout << a[k] << endl;
      break;
    }

  // dosli smo do kraja i nismo nasli ponovljeni
  if (k == 0)
    cout << "nema" << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Двоструки студент

На списку студената који су положили испит један студент је грешком уписан два пута. Напиши програм који проналази тог студента.

Улаз: Са стандардног улаза се уносе имена и презимена студената (њих највише 50 000, свако у посебном реду, раздвојени са по једним размаком).

Излаз: На стандардни излаз исписати име и презиме студента који се два пута јавља на списку.

Пример

Улаз	*Излаз*
pera peric ana anic pera peric mika mikic	pera peric

Задатак: Неупарени елемент

Домаћица матица је на журку позвала своје пријатеље, брачне парове пчеле и трутове. Пошто је гостију пуно, свако је добио број столице. Брачни парови су добили исте бројеве. Који број је добила матица?

Улаз: Са стандардног улаза уноси се број n, а затим и n природних бројева, сваки у посебном реду, од којих се сви осим једног јављају тачно два пута.

Излаз: На стандардни излаз исписати један број - онај који се на улазу јавио тачно једном.

Пример

Улаз	*Излаз*
9 3 2 1 4 2 5 5 3 1	4

Задатак: Број различитих дужина дужи

Дато је N парова тачака које представљају крајеве дужи у простору. Исписати колико различитих дужина дужи се појављује у задатом скупу дужи.

Улаз: У првом реду улаза налази се природан број N ($N \leq 50000$) који представља број дужи. У следећих N редова следи опис тих N дужи са 6 целих бројева ($-10^9 \leq X_1, Y_1, Z_1, X_2, Y_2, Z_2 \leq 10^9$) одвојених празним местима, који редом представљају крајеве сваке дужи.

Излаз: У једини ред излаза потребно је исписати колико различитих дужина се појављује у задатом скупу дужи.

Пример 1

Улаз	*Излаз*
7 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 1 1 1	3

Задатак: Најбројнији елемент

Ученици су гласали за председника одељенске заједнице. Напиши програм који одређује колико гласова је добио победник.

Улаз: Са стандардног улаза се учитава укупан број гласова n ($1 \leq n \leq 10^5$), а затим у наредних n редова по једно име састављено од највише 20 слова енглеске абецеде.

Излаз: На стадардни излаз исписати број гласова који је добио победник (оно име које се најчшеће појавило на улазу).

Пример

Улаз	*Излаз*
5 pera mika jovana pera ana	2

2.8.5 Груписање блиских вредности

Још једна примена сортирања долази од тога што се након сортирања низа, елементи блиски по вредности нађу један близу другог. Ово нам омогућава да у низу налазимо што ближе елементе као и групе што блискијих елемената (са што мањом разликом између најмањег и највећег елемента у групи).

Задатак: Праведна подела чоколадица

Дато је n пакета чоколаде и за сваки од њих је познато колико чоколадица садржи. Сваки од k ученика узима тачно један пакет, при чему је циљ да сви ученици имају што приближнији број чоколадица. Колика је најмања могућа разлика између оног ученика који узме пакет са најмање и оног који узме пакет са највише чоколадица.

Улаз: Са стандардног улаза се уноси природан број n ($1 \leq n \leq 50000$) а затим и n природних бројева (између 1 и 10^6, раздвојене са по једним размаком) који представљају број чоколадица у сваком пакету. У последњем реду се уноси број деце k ($1 \leq k \leq n$).

Излаз: На стандардни излаз исписати вредност најмање разлике.

Пример

Улаз	*Излаз*
8 3 8 1 17 4 7 12 9 4	5

Решење

Сортирање

Најдиректнији начин да се реши задатак је да се испитају сви подскупови од k елемената скупа од n елемената и да се међу њима одабере најбољи. Ово решење је релативно компликовано имплементирати, а уз то је и веома неефикасно (број подскупова је $\binom{n}{k}$, што је $O(n^k)$).

Боље и ефикасније решење се заснива на сортирању. Наиме, када се полазни пакети сортирају по броју чоколадица, ученици треба да узму узастопних k пакета. Претпоставимо да након сортирања имамо низ $a_0, a_1, \ldots, a_{n-1}$. Ученици треба да узму редом пакете од a_i, до a_{i+k-1}, за неко $0 \leq i \leq n-k$.

Докажимо претходну чињеницу и формално. Претпоставимо супротно, да пакети који дају најмањи распон не чине узастопан низ и да сваки узастопни низ пакета дужине k има строго већи распон од распона скупа узетих пакета. Нека је први узети пакет a_i. Тада сигурно постоји неки пакет a_j за $i < j < i+k$ који није узет, а уместо њега је узет неки пакет $a_{j'}$ за неко $i+k \leq j' < n$. Нека је j' последњи пакет који је узет. Међутим, када би ученик који је узео пакет $a_{j'}$ заменио тај пакет за a_j распон би се сигурно смањио или бар остао исти (јер би последњи узети пакет тада био неки пакет $a_{j''}$, за $j'' < j'$, а пошто је низ сортиран неопадајуће, важи да је $a_{j''} \leq a_j$, па и $a_{j''} - a_i \leq a_{j'} - a_i$. Даљим заменама истог типа можемо доћи до тога да су сви узети пакети узастопни, а да је распон мањи или једнак полазном, што је у контрадикцији са тим да је распон полазног скупа узетих пакета строго мањи од распона било којег низа k узастопних пакета.

Разматрање претходног типа је карактеристично за такозване грамзиве алгоритме, о ком ће више речи бити касније.

На основу претходног јасно је да низ бројева чоколадица у пакетима треба најпре сортирати (најбоље помоћу библиотечке функције `sort` и затим одредити минимум разлика вредности $a_{i+k-1} - a_i$, за $0 \leq i \leq n-k$, коришћењем уобичајеног алгоритма за налажење минимума. Разни начини сортирања су описани у задатку Сортирање бројева, док је алгоритам за налажење минимума описан у задатку Најнижа температура.

Сложеношћу овог алгоритма доминира сложеност корака сортирања, а она је $O(n \log n)$, ако се користе библиотечке имплементације. Након сортирања, минимум се одређује у $n-k$ корака, тј. у линеарној сложености $O(n)$ (када је k мало, број корака може бити веома близак n).

```
#include <iostream>
#include <limits>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  // ucitavamo brojeve cokoladica u paketima
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  // ucitavamo broj dece
  int k;
  cin >> k;

  // sortiramo pakete po broju cokoladica
  sort(begin(a), end(a));

  // odredjujemo i ispisujemo najmanju mogucu razliku za nekih k
```

```
  // odabranih paketa
  int min = numeric_limits<int>::max();
  for (int i = 0; i + k - 1 < n; i++) {
    int razlika = a[i + k - 1] - a[i];
    if (razlika < min)
      min = razlika;
  }
  cout << min << endl;

  return 0;
}
```

Задатак: Најближе собе

Два госта су дошла у хотел и желе да одседну у собама које су што ближе једна другој, да би током вечери могли да заједно раде у једној од тих соба. Ако постоји више таквих соба, они бирају да буду што даље од рецепције, тј. у собама са што већим редним бројевима, како им бука не би сметала. Ако је познат списак слободних соба у том тренутку, напиши програм који одређује бројеве соба које гости треба да добију.

Улаз: У првој линији стандардног улаза налази се број n ($1 \leq n \leq 10^5$), а затим се у наредним линијама налазе бројеви слободних соба (у свакој линији по један) - сви бројеви су различити, али је њихов редослед произвољан.

Излаз: На стандардни излаз исписати бројеве соба гостију (прво мањи број, па већи), раздвојене једним размаком.

Пример

Улаз	*Излаз*
7 18 6 25 11 4 1 16	16 18

Задатак: Најбројнији подскуп који садржи узастопне целе бројеве

У низу целих бројева одредити најбројнији подскуп елемената који се могу уредити у низ узастопних целих бројева. На пример, за низ $4, 8, 1, -6, 9, 5, -9, 10, -1, 3, 0, 1, 2$ треба приказати $-1, 0, 1, 2, 3, 4, 5$. Ако има више таквих подскупова, приказати први (онај у којем су бројеви најмањи).

Улаз: У првој линији стандардног улаза уноси се број елемената низа n ($1 \leq n \leq 50000$), а затим у следећих n линија целобројни елементи низа $-100000 \leq a_i \leq 100000$.

Излаз: Најбројнији подскуп узастопних целих бројева (елемената датог низа) уређен у неопадајућем поретку.

Пример

Улаз	*Излаз*
13 4 8 1 -6 9 5 -9 10 -1 3 0 1 2	-1 0 1 2 3 4 5

2.8.6 Свођење на канонски облик

Често имамо потребу да проверимо да ли су два низа елемената једнака, ако се занемари у ком су редоследу елементи наведени. Класичан пример овога је провера да ли се једна реч може добити пермутовањем слова

друге (тј. да ли су анаграми). У суштини, ради се о поређењу два мутлискупа елеманата (која су задата низовима). Најбољи начин да се оваква једнакост провери је да се оба низа сведу на неки канонски облик, који неће зависити од редоследа елемената низа. Најједноставнији начин да се такав канонски облик добије је да се елементи низа сортирају пре поређења. Други начин је да се изврши пребројавање свих елемената тј. да се мултискупови представе пресликавањима сваког елемента у његов број појављивања (два таква пресликавања су једнака ако и само ако исти скуп кључева слика у исте вредности).

Задатак: Анаграм

Дате су две ниске сачињене од малих слова енглеске абецеде, интерпункцијских знакова и размака. Напиши програм који проверава да ли су два дата стринга анаграми, тј. да ли се од прве ниске премештањем слова може добити други друга ниска и обрнуто (карактери који нису слова се занемарују).

Улаз: Прва линија станадардног улаза садржи једну ниску, а друга линија садржи другу ниску.

Излаз: На стандардном излазу у једној линији приказати реч `da` ако дате ниске представљају анаграме, у супротном приказати реч `ne`.

Пример1		**Пример2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`panta redovno zakasni` `neopravdan izostanak`	`da`	`oni su skrsili vagu` `suvisni kilograми`	`ne`

Решење

Две ниске су анаграми ако се једна може добити пермутовањем редоследа карактера друге, па је централни задатак одредити да ли једна ниска представља пермутацију друге. Провера да ли је један низ пермутација другога описана је у задатку Провера пермутација и у овом задатку се могу применити све технике које су у том задатку описане. Једина разлика у односу на обичну проверу пермутација је то што се приликом провере анаграма не узимају у обзир сви карактери, већ само мала слова.

Сортирање

Један начин да се задатак реши је да се обе ниске сортирају и онда упореде. Сортирање ниски може да се изврши библиотечком функцијом. У језику C++ то је функција `sort`. Међутим, пошто се проверавају само слова, пре сортирања и поређења, из ниски је потребно избацити све карактере који нису слова. У језику C++ то је могуће урадити функцијом `copy_if`. Пошто не знамо унапред колико има малих слова, ниску која садржи резултат ћемо проширивати додавањем елемената на крај (за шта користимо `back_inserter`).

Сложеност овог алгоритма зависи од сложености сортирања и ако се користи библиотечка функција сортирања износи $O(n \log n)$. Наиме, копирање ниски (које је неопходно, да се сортирањем не би покварила оригиналне ниске, али и да би се филтрирала само мала слова) и поређење једнакости након сортирања су линеарне сложености, па сортирање узима највише времена.

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

bool anagram(const string& s1, const string& s2) {
  string slova1, slova2;
  copy_if(begin(s1), end(s1), back_inserter(slova1), ::islower);
  copy_if(begin(s2), end(s2), back_inserter(slova2), ::islower);
  sort(begin(slova1), end(slova1));
  sort(begin(slova2), end(slova2));
  return slova1 == slova2;
}

int main() {
  string s1, s2;
  getline(cin, s1);
  getline(cin, s2);
  if (anagram(s1, s2))
```

```
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Провера пермутација

Ленка је добила задатак да испрограмира функцију која "меша" дати низ бројева, тј. одређује његову насумичну пермутацију. Ленка је написала своју функцију, покренула је на одређеном броју тест-примера, међутим, када је добила излазне резултате није одмах могла да види да ли је њена функција исправна. Напиши програм који јој помаже тако што учитава почетни низ елемената и низ добијен мешањем и проверава да ли је други низ пермутација првог тј. да ли се могао добити од првог само променом редоследа његових елемената.

Улаз: Са стандардног улаза се уносе два низа природних бројева. За сваки низ се уноси број елемената (највише 50000), а затим и елементи, сваки у посебном реду.

Излаз: На стандардни излаз испиши реч `da` ако је други низ добијен мешањем првог, тј. `ne` ако није.

Пример

Улаз	*Излаз*
5 1 3 2 4 3 5 4 3 2 3 1	da

Задатак: D-пермутација

Дат је низ који садржи природне бројеве и цео број D. Напиши програм који одређује који се од неколико датих низова може од полазног добити размена елемената који су тачно на растојању D.

Улаз: Са стандардног улаза се учитава цео број d ($1 \leq d \leq k$), затим цео број k ($5 \leq k \leq 10^5$), затим цео број n ($1 \leq n \leq 10$) и након тога n низова дужине k чији су елементи раздвојени размацима.

Излаз: На стандардни излаз за сваки низ од другог до последњег исписати `da` ако се може трансформисати у први размена елемената на растојању d, тј. `ne` у супротном.

Пример

Улаз	*Излаз*
2 7 6 1 2 3 4 5 6 7 3 4 1 2 7 6 5 2 1 4 3 6 5 7 1 4 7 2 3 6 5 1 4 7 6 3 2 5 5 4 7 6 3 1 2	da ne da da ne

Објашњење

```
3 4 1 2 7 6 5   - razmena 3 i 1, 4 i 2, i 7 i 5
```

```
2 1 4 3 6 5 7   - ne može
1 4 7 2 3 6 5   - razmena 4 i 2, 7 i 3 i 7 i 5
1 4 7 6 3 2 5   - razmena 6 i 2, 4 i 2, 7 i 3 i 7 i 5
5 4 7 6 3 1 2   - ne može
```

Решење

Сортирање

Приметимо да се елемент на позицији 0, након пермутације може наћи само на позицији d, $2d$, $3d$ итд. Елемент на позицији 1 се може наћи само на позицијама $d+1$, $2d+1$, $3d+1$ итд. На крају, елемент на позицији $d-1$ се може наћи само на позицијама $d+d-1$, $2d+d-1$, $3d+d-1$ итд.

На пример, у низу, 1, 2, 3, 4, 5, 6, 7, 8, 9, ако је растојање d једнако 3, можемо размењивати само елементе 1, 4 и 7, затим елементе 2, 5 и 8 и на крају 3, 6 и 9. Не постоји никакав начин да се, на пример, елемент 2 или елемент 9 нађу на позицији елемента 1 или елемента 7.

Дакле, низ је суштински издељен на d партиција, тако што су два елемента у истој партицији ако и само ако су на растојању $i \cdot d$, за неко целобројно i. Партиције су међусобно независне у смислу да се размена могу пермутовати само они елементи који се налазе у истој партицији. Један низ се може трансформисати у други размена елемената на растојању d, ако и само ако се свака партиција може трансформисати у одговарајућу партицију другог низа, размена суседних елемената у оквиру те партиције.

На пример, низ 1, 2, 3, 4, 5, 6, 7, 8, 9 се може трансформисати у низ 7, 2, 9, 1, 8, 6, 4, 5, 3, зато што се партиција $(1, 4, 7)$ може трансформисати у $(7, 1, 4)$, партиција $(2, 5, 8)$ се може трансформисати у $(2, 8, 5)$, а партиција $(3, 6, 9)$ се може трансформисати у партицију $(9, 6, 3)$. Пошто свака партиција чини низ за себе, остаје да се реши питање: да ли се дати низ бројева може трансформисати у други низ операцијама размене суседних елемената?

Постоје разни начини да се на ово питање одговори (слично као у задатку Провера пермутација), а један од најефикаснијих се заснива на чињеници да се сваки низ може сортирати искључиво размена узастопних елемената (на пример, Bubble Sort алгоритам врши управо такво сортирање). То значи да се један низ трансформацијама узастопних елемената може трансформисати у други низ ако и само ако се сортирањем и једног и другог добија исти низ. При том, сортирање не мора бити вршено само размена узастопних елемената већ и много ефикаснијим алгоритмима (јер је резултат сортирања јединствено одређен и ако знамо да се неким ефикасним алгоритмом сортирања од почетног низа добија неки сортиран низ, тај исти низ би се добио и да се сортирање врши само размена узастопних елемената). Један од начина да претходна разматрања искомбинујемо у решење је да уведемо операцију проналажења канонског представника за сваки низ у односу на дати параметар d, тако што ћемо сваку његову партицију канонизовати тј. сортирати.

На пример, за $d = 3$ низ $2, 16, 41, 15, 12, 31, 11, 9$ се дели на партиције $(2, 15, 11)$, $(16, 12, 9)$, $(41, 31)$, свака се партиција се независно сортира и тиме се добија $(2, 11, 15)$, $(9, 12, 16)$, и $(31, 41)$, чиме се добија канонски представник $2, 9, 31, 11, 12, 41, 15, 16$. Два низа ће се моћи d-пермутовати један у други ако и само ако су њихови овако дефинисани канонски представници једнаки.

Пошто у оквиру канонизације користимо библиотечки алгоритам сортирања можемо претпоставити да ће његова сложеност за низ дужине m бити $O(m \cdot \log(m))$. Пошто се сортира d партиција, а свака је дужине око k/d, и врши се пребацивање елемената у помоћни низ и назад, сложеност најгорег случаја канонизације се може проценити као $O(d \cdot (k/d \cdot \log(k/d) + 2(k/d)))$ тј. $O(k \cdot \log(k/d))$. Зато израчунавање канонских представника свих низова може да се процени на $O(n \cdot k \cdot \log(k/d))$. С обзиром на то да се поређење једнакости два канонска представника врши у времену $O(k)$, а ово поређење се врши $n-1$ пут, имамо додатно још $O(nk)$ операција. За укупну сложеност алгоритма се зато може узети $O(n \cdot k \cdot \log(k/d))$, тј. $O(k \cdot \log(k/d))$ јер n, за разлику од k има веома малу вредност (практично се може сматрати константом).

Напоменимо да је након канонизације сваке партиције низа T_i било могуће проверавати једнакост са одговарајућом партицијом низа S, чиме би се мало брже могло констатовати да се низ не може трансформисати, али би се код донекле закомпликовао. Слично, коришћење помоћног низа није било неопходно за чување партиција, већ је било могуће канонизацију вршити у оквиру низа S, размена његових елемената из истих партиција, али тада не би било могуће користити библиотечку функцију сортирања, што би такође доста закомпликовало код. Асимптотска сложеност најгорег случаја би остала иста.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```cpp
using namespace std;

// određivanje kanonskog predstavnika za niz s
void kanonizuj(vector<int>& s, int d) {
  int duzina = s.size();
  // pomocni vektor koji ce cuvati particiju po particiju
  vector<int> particija(duzina / d + 1);

  // kanonizovacemo svaku od d particija
  for (int i = 0; i < d; i++) {
    int m;

    // prebacujemo elemente i-te particije u pomocni vektor
    m = 0;
    for (int j = i; j < duzina; j += d)
      particija[m++] = s[j];

    // sortiramo elemente particije unutar pomocnog vektora
    sort(particija.begin(), particija.begin() + m);

    // vracamo elemente particije iz pomocnog vektora nazad u niz s
    m = 0;
    for (int j = i; j < duzina; j += d)
      s[j] = particija[m++];
  }
}


int main() {
  // iskljucujemo sinhronizaciju da bi ucitavanje teklo brze
  ios::sync_with_stdio(false);

  // ucitavamo rastojanje d
  int d;
  cin >> d;
  // ucitavamo dužinu nizova k
  int k;
  cin >> k;
  // ucitavamo broj nizova
  int n;
  cin >> n;

  // ucitavamo niz S
  vector<int> s(k);
  for (int i = 0; i < k; i++)
    cin >> s[i];

  // odredjujemo kanonskog predstavnika niza s
  kanonizuj(s, d);

  // obradjujemo preostalih n-1 nizova Ti
  for (int j = 1; j < n; j++) {
    // ucitvamo niz Ti
    vector<int> t(k);
    for (int i = 0; i < k; i++)
      cin >> t[i];
```

```
    // odredjujemo kanonskog predstavnika niza Ti
    kanonizuj(t, d);

    // odredjujemo da li se niz S moze transformisati u niz Ti
    // (moze ako i samo ako su im kanonski predstavnici jednaki)
    // i ispisujemo rezultat u trazenom formatu
    cout << (s == t ? "da" : "ne") << endl;
  }

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Анаграми

Две речи су анаграми ако се једна може добити од друге само променом редоследа слова (на пример, трава и ватра). Напиши програм који у датом скупу речи проналази највећи број речи које су међусобни анаграми.

Улаз: Са стандардног улаза се уноси број речи n ($0 \leq n \leq 50000$), а затим у n наредних линија по једна реч полазног скупа (све речи се састоје само од малих слова енглеског алфабета и имају највише 200 карактера).

Излаз: На стандардни излаз исписати тражену величину највећег подскупа полазног скупа речи, у коме су све речи једна другој анаграми.

Пример

Улаз	*Излаз*
10 tommarvoloriddle viviandarkbloom iamlordvoldemort danabnormal normdanabal vladimirnabokov vladimirkoborov bladvakvinomori damonalbarn dorianvivalkomb	4

Решење

О техникама провере да ли су две речи анаграми било је речи у задатку Анаграм. Провера се може заснивати било на сортирању слова две речи и затим поређењу да ли су добијене речи исте, било на поређењу броја појављивања сваког слова. Дакле, сваку реч можемо нормализовати или тако што је сортирамо или тако што израчунамо број појављивања сваког од 26 карактера енглеске абецеде. Тиме добијамо низ канонских представника сваке речи и желимо да у том низу пронађемо скуп еквивалентних елемената (једнаких канонских представника) који је највеће кардиналности од свих таквих подскупова. То је веома слично задатку Фреквенције речи и опет се може решити било сортирањем, па тражењем најдуже серије једнаких узастопних елемената (као у задатку Најдужа серија победа или Најдужа растућа серија), било бројањем појављивања сваког канонског представника.

Сортирање слова, па сортирање речи

Дакле, једно могуће решење је да сортирамо карактере сваке учитане речи (абецедно), а затим да сортирамо тако добијени низ речи (лексикографски абецедно), да би се идентичне сортиране речи нашле једна иза друге. На тако добијен низ примењујемо проналажење најдуже серије једнаких узастопних елемената.

Сложеност сортирања слова сваке речи је $O(n \cdot m \log m)$, где је n број речи, а m максимални број слова. Сложеност сортирања свих речи се може проценити на $O(n \log n)$, јер је реално очекивати да ће се лексикографско поређење две речи вршити веома ефикасно (речи су кратке, а реално је очекивати и да су "шаренолике", тј. да ће се њихов поредак моћи одредити већ након поређења малог броја почетних карактера). Пошто је број слова m веома мали, можемо га сматрати константим и сложеност алгоритма грубо оценити са $O(n \log n)$.

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<string> reci(n);
  for (int i = 0; i < n; i++)
    cin >> reci[i];

  // sortiramo karaktere svake reci
  for (int i = 0; i < n; i++)
    sort(begin(reci[i]), end(reci[i]));
  // sortiramo ceo niz reci leksikografski
  sort(begin(reci), end(reci));

  // odredjujemo duzinu najduze serije jednakih reci
  int maxDuzinaSerije = 1;
  int tekucaDuzinaSerije = 1;
  for (int i = 1; i < n; i++) {
    if (reci[i] == reci[i-1])
      tekucaDuzinaSerije++;
    else
      tekucaDuzinaSerije = 1;
    if (tekucaDuzinaSerije > maxDuzinaSerije)
      maxDuzinaSerije = tekucaDuzinaSerije;
  }

  cout << maxDuzinaSerije << endl;

  return 0;
}
```

Бројање свих слова, па сортирање низова бројача

Једно могуће решење је да за сваку реч формирамо низ који садржи број појављивања сваког слова. Број појављивања пресликава свако слово у његов број појављивања, а може се једноставно имплементирати помоћу обичног низа (уместо мапе или речника), како је показано у задатку Фреквенција знака.

Када се за сваку реч направи низ бројева појављивања сваког слова, добијамо низ низова који онда сортирамо да би се идентични низови нашли један иза другога (поредећи тако добијене низове бројева лексикографски). На тако сортирани низ примењујемо проналажење најдуже серије једнаких узастопних елемената.

Формирање низова бројача за сваку реч захтева $O(nm)$ корака. Након тога, сортирање низова бројача захтева $O(n \log n)$ корака (при чему смо претпоставили да се поређење два низа бројача извршава практично у константној сложености, јер је бројача тек 26, а и реално је очекивати да се поредак два низа бројача може установити тек поређењем неколико бројача). Пошто су речи прилично кратке, сложеност алгоритма се грубо може проценити са $O(n \log n)$.

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;
```

```
vector<int> prebrojSlova(const string& s) {
  vector<int> rez(26);
  for (char c : s)
    rez[c - 'a']++;
  return rez;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<vector<int>> broj_slova(n);
  for (int i = 0; i < n; i++) {
    string s;
    cin >> s;
    broj_slova[i] = prebrojSlova(s);
  }

  sort(begin(broj_slova), end(broj_slova));

  // odredjujemo duzinu najduze serije jednakih reci
  int maxDuzinaSerije = 1;
  int tekucaDuzinaSerije = 1;
  for (int i = 1; i < n; i++) {
    if (broj_slova[i] == broj_slova[i-1])
      tekucaDuzinaSerije++;
    else
      tekucaDuzinaSerije = 1;
    if (tekucaDuzinaSerije > maxDuzinaSerije)
      maxDuzinaSerije = tekucaDuzinaSerije;
  }

  cout << maxDuzinaSerije << endl;
}
```

Види другачија решења овог задатка.

2.8.7 Сортирање интервала

Многи проблеми се природно моделују интервалима (на пример, време доласка и време одласка неке особе на посао, заузеће неке сале и слично). Многи алгоритми који захтевају обраду неке колекције интервала се ефикасније реализују ако се ти интервали обилазе у неком редоследу. То може бити редослед одређен на основу почетка интервала, на основу краја интервала, а често и редослед који редом обилази све значајне тачке интервала (почетке и крајеве интервала) - у овом случају се не обилазе сами интервали по неком реду, већ само њихове значајне тачке.

Задатак: Најбројнији пресек интервала

Људи су долазили и одлазили са базена и за сваког посетиоца је познато време доласка и време одласка. Претпоставићемо да је човек на базену у периоду облика $[a, b)$, тј. да се човек налази на базену у тренутку свог доласка a, а да се не налази у тренутку свог одласка b. Колико је људи највише било истовремено на базену?

Улаз: Са стандардног улаза се учитава број посетилаца n ($1 \leq n \leq 50000$), а затим у наредних n редова време доласка и време одласка сваког посетиоца (мерење је веома прецизно, па се време представља природним бројевима мањим од милијарде), одвојене са по једним размаком.

Излаз: На стандардни излаз исписати тражени максимални број посетилаца у неком тренутку.

Пример

Улаз *Излаз*

```
8        5
3 7
7 8
2 5
6 8
4 6
1 6
4 5
1 2
```

Објашњење

```
    1 2 3 4 5 6 7 8
3 7     x x x x
7 8             x
2 5   x x x
6 8           x x
4 6       x x
1 6   x x x x x
4 5       x
1 2   x
```

У тренутку 4 на базену се налази 5 посетилаца.

Решење

Бројач за сваки појединачан тренутак

Директно решење би подразумевало да се одржава низ у коме се за сваки тренутак памти број људи на базену. Пошто не знамо колико тренутака постоји уместо низа можемо употребити хеш-таблицу. У језику C++ можемо употребити `unordered_map`.

Ако је тренутака и људи пуно, ова метода ће бити веома неефикасна. Сложеност најгорег случаја је $O(n \cdot m)$, где је n број тренутака, а m број људи који су тај дан посетили базен.

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main() {
  int n;
  cin >> n;

  unordered_map<int, int> brojPosetilaca;
  for (int i = 0; i < n; i++) {
    int dosao, otisao;
    cin >> dosao >> otisao;
    for (int j = dosao; j < otisao; j++)
      brojPosetilaca[j]++;
  }

  int maksPrisutno = 0;
  for (auto it : brojPosetilaca)
    if (it.second > maksPrisutno)
      maksPrisutno = it.second;

  cout << maksPrisutno << endl;

  return 0;
}
```

Сортирани низ карактеристичних тренутака (долазака и одлазака)

Број људи који су тренутно на базену мења се само у тренуцима када неко дође или када неко оде. Да би се одредио највећи број људи довољно је размотрити само те карактеристичне тренутке. Веома природно је да те карактеристичне тренутке обрађујемо хронолошки, у растућем редоследу времена. Можемо креирати низ који садржи све карактеристичне тренутке (времена одласка и доласка) и за сваки тренутак бележити да ли је долазни или одлазни тренутак. Тај низ можемо сортирати и затим обрађивати редом, израчунавајући за сваки тренутак број људи на базену инкрементално, на основу броја људи на базену у претходном карактеристичном тренутку. Ако у неком временском тренутку више људи долази или одлази, број људи ћемо упоређивати са максимумом тек када обрадимо све људе који су дошли или отишли у том тренутку (што решавамо тако што у унутрашњој петљи обрађујемо све људе који су дошли и отишли у текућем тренутку).

Ако постоји n људи који су били на базену, постоји $2n$ карактеристичних тренутака за чије је сортирање потребно $O(n \log n)$ корака. Након сортирања, низ тренутака се обрађује једним проласком кроз низ у линеарном времену (обратите пажњу на то да иако у имплементацији постоје две угнежђене петље, променљива `i` само увећава своју вредност и укупно се кроз обе петље изврши највише $2n$ корака). Дакле, сложеношћу доминира сортирање и сложеност овог решења је $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

  // niz karakterističnih trenutaka
  vector<pair<int, int>> promene(2*n);
  for (int i = 0; i < n; i++) {
    int dosao, otisao;
    cin >> dosao >> otisao;
    promene[2*i] = make_pair(dosao, 1);
    promene[2*i+1] = make_pair(otisao, -1);
  }

  sort(begin(promene), end(promene));

  int trenutnoPrisutno = 0;
  int maksPrisutno = 0;
  int i = 0;
  while (i < 2*n) {
    int trenutak = promene[i].first;
    while (i < 2*n && promene[i].first == trenutak)
      trenutnoPrisutno += promene[i++].second;
    if (trenutnoPrisutno > maksPrisutno)
      maksPrisutno = trenutnoPrisutno;
  }

  cout << maksPrisutno << endl;

  return 0;
}
```

Елиминисање унутрашње петље

Ако се сортирање парова ради лексикографски (прво по времену, а онда по ознаци 1 тј. -1), тако да су сви догађаји који су се десили у истом тренутку сортирани тако да прво иду они који су отишли (ознака -1), па онда они који су дошли (ознака 1), тада не морамо да имамо унутрашњу петљу, јер ће се број прво смањивати,

па онда расти и неће бити могуће да се добије погрешан резултат зато што су додати неки посетиоци пре него што је констатовано да су неки отишли.

Сложеност решења остаје $O(n \log n)$, док се у овој имплементацији још јасније види да је фаза након сортирања низа линеарне сложености тј. $O(n)$.

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

  // niz karakterističnih trenutaka
  vector<pair<int, int>> promene(2*n);
  for (int i = 0; i < n; i++) {
    int dosao, otisao;
    cin >> dosao >> otisao;
    promene[2*i] = make_pair(dosao, 1);
    promene[2*i+1] = make_pair(otisao, -1);
  }

  sort(begin(promene), end(promene));

  int trenutnoPrisutno = 0;
  int maksPrisutno = 0;
  for (int i = 0; i < 2*n; i++) {
    trenutnoPrisutno += promene[i].second;
    if (trenutnoPrisutno > maksPrisutno)
      maksPrisutno = trenutnoPrisutno;
  }

  cout << maksPrisutno << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Покривање праве затвореним интервалима

Написати програм који за низ затворених интервала $[a_i, b_i], i = 0, 1, \cdots, n-1$ одређује дужину дела праве коју покривају и минимални број интервала којим се може постићи покривање истог скупа тачака праве (тај скуп интервала може се добити укрупњивањем полазних интервала, тј. обједињавањем полазних интервала који се секу).

Улаз: У првој линији стандардног улаза учитава се број интервала n ($1 \leq n \leq 50000$), а у наредних n линија парови целих бројева за које важи $-10^6 \leq L_i < R_i \leq 10^6$ који представљају леви и десни крај интервала.

Излаз: Два броја: у првом реду стандардног излаза број који представља дужину дела праве коју покривају учитани интервали, у следећем реду број интервала формираних укрупњавањем међусобно повезаних интервала.

Пример

Улаз	*Излаз*
3 1 3 5 8 2 4	6 2

Објашњење

интервали $[1, 3]$ и $[2, 4]$ се могу укрупнити у интервал $[1, 4]$, док интервал $[5, 8]$ остаје непромењен.

2.8.8 Остале примене сортирања

Наведени примери сортирања сигурно не исцрпљују све употребе сортирања. У наставку ће бити приказано још неколико задатака који могу ефикасно бити решени захваљујући примени сортирања.

Задатак: Хиршов h-индекс

Рангирање научника врши се помоћу статистике која се назива *Хиршов индекс* (скр. *h-индекс*). H-индекс научника је највећи број h такав да научник има бар h радова од којих сваки има бар h цитата.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 5 \cdot 10^4$) који представља број радова научника а затим и n природних бројева који представљају број цитата (између 0 и 10^6) за сваки од тих n радова.

Излаз: На стандардни излаз исписати један природан број који представља h-индекс научника.

Пример

Улаз	*Излаз*
8 3 5 12 7 5 9 0 17	5

Објашњење

Постоји тачно 5 радова са бар 5 цитата (5, 12, 9, 7, 17). Преостали радови имају 3, 5 и 0 цитата, тако да не постоји 6 радова са бар 6 цитата.

Види другачија решења овог задатка.

Задатак: Збир минимума тројки

Дат је низ позитивних целих бројева $a_0, a_1, ..., a_{n-1}$. За сваку тројку $0 \leq i < j < k \leq n$ одредити најмању вредност од три броја a_i, a_j, a_k, а затим одредити последњих 6 цифара збира тако добијених вредности.

Улаз: У првој линији стандардног улаза уноси се број елемената низа n ($1 \leq n \leq 10000$), а затим у следећих n линија елементи низа $1 \leq a_i \leq 100000$.

Излаз: Последњих 6 цифара збира.

Пример 1

Улаз	*Излаз*
5 3 8 4 2 6	25

Пример 2

Улаз	*Излаз*
10 909111 56723 73737 636 91919 10000 23400 6258 10002 67654	409187

Задатак: Коректни телефони

Низ телефонских бројева је коректан ако ниједан број није префикс другога (самим тим у њему не постоје ни два иста броја). Напиши програм који одређује да ли је низ унетих бројева коректан.

Улаз: Са стандардног улаза се уноси број бројева n ($1 \leq n \leq 50000$), затим у n наредних линија n бројева. Сваки број се састоји од између 3 и 50 цифара.

Излаз: На стандардни излаз исписати `da` ако је низ коректан тј. `ne` ако није.

Пример 1

Улаз	*Излаз*
4	ne
192	
194	
199342	
192865	

Пример 2

Улаз	*Излаз*
4	da
199342	
193865	
192	
194	

Решење

Задатак можемо решити и елементарније, тако што ћемо прво бројеве сортирати лексикографски, а затим ћемо проверити да ли се међу паровима узастопних бројева налази неки у коме је први број префикс другог.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<string> numbers(n);
  for (int i = 0; i < n; i++)
    cin >> numbers[i];
  sort(begin(numbers), end(numbers));
  bool OK = true;
  for (size_t i = 1; i < numbers.size(); i++)
    if (numbers[i-1].size() <= numbers[i].size() &&
        equal(begin(numbers[i-1]), end(numbers[i-1]), begin(numbers[i]))) {
      OK = false;
      break;
    }
  if (OK)
    cout << "da" << endl;
  else
    cout << "ne" << endl;

  return 0;
}
```

2.9 Бинарна претрага

Ако немамо никакве информације о редоследу елемената у низу, једини начин да проверимо да ли се у њему налази неки елемент је да употребимо линеарну претрагу и да редом проверавамо један по један елемент низа. У најгорем случају сваки елемент низа мора бити прегледан, па је сложеност таквог приступа $O(n)$, где је n број елемената низа.

Ако је низ елемената сортиран, претрагу је могуће извршити много ефикасније, у сложености $O(\log n)$, коришћењем алгоритма **бинарне претраге**, којим се, услед сортираности низа, одсеца значајан део простора претраге и тако добија на ефикасности. Она се може применити на много сродних проблема.

У основној варијанти, бинарна претрага (БП) служи да се провери да ли сортирани низ елемената садржи неку дату вредност.

Поред овога, БП се може употребити да се пронађе први елемент у сортираном низу који је (било строго, било нестрого) већи или мањи од датог.

У свом најопштијем облику бинарна претрага се користи да се у низу пронађе преломна тачка тј. први елемент који задовољава неки услов (под претпоставком да су елементи поређани тако да у низу прво иду сви елементи који тај услов не задовољавају, а затим они који тај услов задовољавају).

Бинарну претрагу ћемо употребљавати и за оптимизацију, тј. да пронађемо најмању или највећу вредност, која задовољава одређени услов.

2.9.1 Бинарна претрага елемента у низу

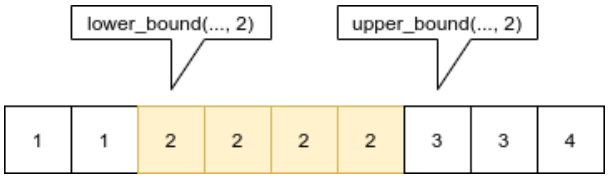
Алгоритам бинарне претраге вредности у низу одговара оном који се може применити у игри погађања непознатог броја и који је заснован на половљењу интервала. Основна идеја је да се тражени елемент пореди са средишњим елементом у низу. Ако је тражени елемент мањи од средишњег, пошто је низ сортиран, знаћемо да је мањи и од свих елемената десно од тог средишњег, па тај део низа можемо елиминисати (можемо начинити одсецање у претрази) и претрагу можемо наставити само у левој половини низа. Симетрично, ако је тражени елемент већи од средишњег, због сортираности је већи и од свих елемената лево од средишњег и лева половина низа може бити елиминисана из даље претраге. На крају, ако елемент није ни мањи ни већи од средишњег, онда му је једнак и пронађен је у низу. Приметимо да у основи алгоритма бинарне претраге лежи техника одсецања, која је оправдана тиме што је низ сортиран.

Пошто се у сваком кораку претраге дужина низа дупло смањује, за претрагу целог низа довољно је $O(\log n)$ корака, где је n дужина низа. Наиме, претрага у најгорем случају траје све док се половљењем низ не испразни. Дужина низа након k корака половљења је отприлике $\frac{n}{2^k}$. Низ ће се испразнити када је $\frac{n}{2^k} < 1$, тј. када је $n < 2^k$, тј. када је $k > \log_2 n$.

Слично као и за сортирање, већина програмских језика пружа готове библиотечке функције за бинарну претрагу.

У језику C++ функција `binary_search` проверава да ли дати распон елемената (задат помоћу два итератора) садржи задату вредност (функција враћа `true` ако и само ако се тражени елемент налази унутар задатог распона). Тако се провера да ли се дати елемент `x` налази унутар сортираног вектора `a` може извршити помоћу `binary_search(begin(a), end(a), x)`.

Поред ове, постоје још три функције које врше одређене варијације бинарне претраге. Ако је потребно пронаћи све елементе једнаке датом, можемо користити функцију `equal_range` (са истим параметрима као `binary_search`). Она враћа пар итератора који ограничавају распон елемената једнаких датом (први итератор указује на први елемент једнак траженој вредности, а други на позицију непосредно иза последњег елемента једнаког траженој вредности). Функција `lower_bound` враћа први од та два итератора (тј. први елемент који је већи или једнак од тражене вредности), а функција `upper_bound` враћа други од њих (тј. први елемент који је строго већи од тражене вредности).



Слика 2.5: lower_bound и upper_bound

О њима и њиховој употреби ће бити више речи у наредним поглављима.

У свим библиотечким функцијама за бинарну претрагу, ако се не зада другачије, подразумева се да је низ сортиран у односу на подразумевани поредак елемената (неопадајући нумерички ако су бројеви у питању, тј. неопадајући абецедни лексикографски ако су ниске у питању). Поредак се може задати или променити на сличан начин као код функција за сортирање (о чему ће бити више речи касније).

Задатак: Провера бар-кодова

У продавници се налази пуно врста производа и познати су њихови бар-кодови. Произвођач жели да сазна колико се врста његових производа продаје у тој продавници. Ако је списак свих кодова производа у продавници дат у сортираном облику, а списак свих кодова производа произвођача је достављен несортиран, напиши програм који одређује тражени број.

Улаз: Са стандардног улаза учитава се број n ($1 \leq n \leq 50000$), а n природних бројева (највише шестоцифрених), раздвојених размацима. Ти бројеви представљају бар-кодове производа у продавници и сортирани су растуће. Након тога се до краја улаза учитавају бар-кодови производа које је произвођач доставио (највише шестоцифрени природни бројеви, сваки у посебном реду).

Излаз: На станадрни излаз исписати број производа произвођача који се већ продају у продавници.

Пример

Улаз	*Излаз*
5 1 3 5 6 7 2 3 4 5 8	2

Решење

Линеарна претрага

Наиван начин да проверимо да ли елемент постоји у низу је примена алгоритма линеарне претраге. Разни начини имплементације линеарне претраге описани су у задатку Негативан број. Линеарна претрага може бити било имплементирана ручно, било реализована помоћу библиотечких функција. У језику C++ функцијом `find` можемо проверити да ли низ садржи дати елемент.

Пошто се у најгорем случају линеарном претрагом (било библиотечком, било ручно-имплементираном) анализира сваки елемент низа, сложеност претраге једног елемента је $O(n)$. Ако претпоставимо да се претражује m елемената, укупна сложеност алгоритма је $O(mn)$.

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

// funkcija proverava da li se u datom sortiranom vektoru a nalazi
// element x
bool sadrzi(const vector<int>& a, int x) {
  // proveravamo sve elemente vektora a
  for (int i = 0; i < a.size(); i++)
    // nasli smo element x na poziciji i
    if (a[i] == x)
      return true;
  // element x nije nadjen
  return false;
}

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // brojac pojavljivanja elemenata
  int broj = 0;
  // ucitavamo element po element do kraja ulaza
  int x;
```

```
  while (cin >> x) {
    // ako je ucitani element sadrzan u nizu a, uvecavamo brojac
    if (sadrzi(a, x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;
  return 0;
}
```

Бинарна претрага

Чињеница да је низ сортиран нам даје начин да задатак решимо много ефикасније, применом бинарне претраге.

Низ од n елемената се бинарном претрагом може претражити у сложености $O(\log n)$, па m производа можемо претражити у сложености $O(m \log n)$.

Бибилотечке функције

У језику C++ функција `binary_search` изводи бинарну претрагу неког сегмента (низа или вектора). Аргументи су два итератора (један на први елемент сегмента који се претражује, а други непосредно иза последњег елемента) и елемент који се тражи. Функција враћа `bool` чија је вредност `true` ако и само ако елемент постоји у низу.

Библиотечка функција гарантује да ће сортиран низ од n елемената бити претражен у сложености $O(\log n)$ (па се m елемената независно претражује у $O(m \log n)$ корака).

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // broj onih koji postoje u nizu
  int broj = 0;
  // ucitavamo broj po broj do kraja ulaza
  int x;
  while (cin >> x) {
    // ako je broj sadrzan u nizu, uvecavamo brojac
    if (binary_search(a.begin(), a.end(), x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;

  return 0;
}
```

Итеративна имплементација

Алгоритам бинарне претраге вредности у низу одговара оном који се може применити у игри погађања непознатог броја и који је заснован на половљењу интервала.

Претпоставимо да желимо проверити да ли се елемент x налази у низу a између позиција l и d тј. да ли се налази у затвореном интервалу позиција $[l, d]$. Ако је интервал празан, он не садржи елемент x. У супротном пронађимо средину овог интервала s.

- Ако је $x < a[s]$, пошто је низ сортиран неопадајуће, важи да је x мањи и од свих елемената који су десно од s. Зато претрагу можемо наставити у интервалу $[l, s-1]$.
- Слично, ако је $x > a[s]$, пошто је низ сортиран неопадајуће, важи да је x веће од свих елемената лево од s тако да претрагу можемо наставити само у интервалу $[s+1, d]$.
- На крају, ако је $x = a[s]$ тада смо елемент пронашли у низу и то на позицији s. Претрага се може прекинути и када се интервал који се претражује испразни (што ће се десити ако је $l > d$).

Алгоритам можемо имплементирати итеративно. Променљиве l и d иницијализујемо на 0 и $n-1$, затим у петљи која се извршава док је $l \leq d$ проналазимо средину s интервала $[l, d]$ и поредимо $a[s]$ са x. Ако је $x < a[s]$ тада d постављамо на $s-1$, ако је $x > a[s]$ тада l постављамо на $s+1$, а ако је $x = a[s]$ прекидамо функцију (и петљу) информишући позиваоца да је елемент нађен на позицији s. Ако се петља заврши, елемент не постоји у низу.

Докажимо формално коректност овог алгоритма. Инваријанта петље је да су:

- сви елементи у интервалу $[0, l)$ строго мањи од x,
- сви елементи у интервалу (d, n) строго већи од x.

Уз то важи и $0 \leq l \leq d+1 \leq n$.

Иницијализацијом $l = 0$ и $d = n-1$, инваријанта је задовољена.

Претпоставимо да инваријанта важи при уласку у петљу, након провере услова $l \leq d$. Тада израчунавамо средину интервала s (за њу сигурно важи $l \leq s \leq d$).

- Ако је $x < a_s$, тада је $l' = l$, $d' = s-1$. Пошто је $a_s > x$, услед сортираности низа у неопадајућем поретку, већи од x су и сви елементи на позицијама из интервала $(d', n) = (s-1, n) = [s, n)$. Пошто се вредност променљиве `l` није променила, сви елементи на позицијама $[0, l')$ су сигурно строго мањи од x (што знамо да важи на основу претпоставке).
- Ако је $x > a_s$, тада је $l' = s+1$, $d = d$. Пошто је $a_s < x$, услед сортираности низа у неопадајућем поретку, мањи од x су и сви елементи на позицијама из интервала $[0, l') = [0, s+1) = [0, s]$. Пошто се вредност променљиве `d` није променила, сви елементи на позицијама (d', n) су сигурно строго већи од x (што знамо да важи на основу претпоставке).
- Ако је $x = a_s$, пронађен је тражени елемент и функција коректно потврђује да низ садржи елемент x.

Када се петља заврши важи инваријанта, али услов $l \leq d$ није испуњен. Пошто је $l \leq d+1$, важи да је $l = d+1$. Стога су сви елементи у интервалу $[0, l)$ строго мањи од x а сви елементи у интервалу $(d, n) = [d+1, n) = [l, n)$ строго већи од x. Дакле, сви елементи низа су или строго мањи или строго већи од x и стога низ не садржи елемент x.

Алгоритам се сигурно зауставља, јер се у сваком кораку петље ширина интервала $[l, d]$, која је једнака $d-l+1$ строго смањује, све док не достигне нулу.

Пошто се у сваком кораку интервал $[l, d]$ полови, пошто иницијално крећемо од интервала $[0, n-1]$ и пошто се претрага завршава када се интервал испразни, сложеност претраге једног елемента је $O(\log n)$ (па се m елемената независно претражује у $O(m \log n)$ корака).

```
#include <iostream>

using namespace std;

// funkcija proverava da li se u datom sortiranom nizu a duzine n
// nalazi element x
bool sadrzi(int a[], int n, int x) {
  // petrazujemo da li se element nalazi u intervalu [l, d]
  int l = 0, d = n - 1;
  // dok god taj interval nije prazan
  while (l <= d) {
```

```
    // nalazimo sredinu intervala
    int s = l + (d - l) / 2;

    // ako je x manji od srednjeg on se moze nalaziti samo u intevalu
    // [a, s-1] (jer je niz sortiran)
    if (x < a[s])
      d = s - 1;
    // ako je x veci od srednjeg on se moze nalaziti samo u intevalu
    // [s+1, d] (jer je niz sortiran)
    else if (x > a[s])
      l = s + 1;
    else
      // nasli smo element x na poziciji s
      return true;
  }
  // element ne postoji u nizu
  return false;
}

// maksimalni broj elemenata niza dat tekstom zadatka
const int MAX = 50000;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  int a[MAX];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // broj onih koji postoje u nizu
  int broj = 0;
  // ucitavamo broj po broj do kraja ulaza
  int x;
  while (cin >> x) {
    // ako je broj sadrzan u nizu, uvecavamo brojac
    if (sadrzi(a, n, x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Број парова датог збира

Дат је цео број s и низ различитих целих бројева. Написати програм којим се одређује број парова у низу који имају збир једнак датом броју s.

Улаз: У првој линији стандардног улаза налази се цео број s (број из интервала $[0, 10^6]$), у другој линији налази се број елемената низа n ($1 \leq n \leq 50000$), а у следећих n линија налази се редом елементи низа (бројеви из интервала $[0, 10^6]$).

Излаз: На стандардном излазу приказати број парова различитих елемената низа чији је збир једнак броју s.

Пример

Улаз	*Излаз*
5 6 1 4 3 6 -1 5	2

Задатак: Квадрати

Дат је скуп тачака у равни. Написати програм који одређује колико се различитих квадрата може направити тако да су им сва четири темена у том скупу тачака.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 1000$), а затим у наредних n линија координате тачака (прво x, онда y, раздвојене једним размаком).

Излаз: На стандардни излаз исписати тражени број квадрата.

Пример

Улаз	*Излаз*
7 -1 1 0 1 1 1 -1 0 0 0 1 0 0 -1	3

Решење

Груба сила

Решење грубом силом је да се провери свака четворка тачака и утврди да ли чини квадрат. Да би се проверило да ли четири тачке $ABCD$ чине квадрат, довољно је да се провери да ли су дужине страница и дужине дијагонала једнаке (проверавамо да ли је $AB = BC$, $BC = CD$, $CD = DA$ и да ли је $AC = BD$). Сваки квадрат ће бити избројан 8 пута (по четири цикличке пермутације темена за сваку од две оријентације), тако да добијени број треба поделити са 8.

Пошто се испитују све четворке тачака, којих има $\binom{n}{4} = \frac{n(n-1)(n-2)(n-3)}{4!}$, а провера да ли оне чине квадрат се врши у константној сложености, сложеност овог приступа је $O(n^4)$, што је веома неефикасно.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Tacka {
  int x, y;
};

int kvadratRastojanja(const Tacka& t1, const Tacka& t2) {
  return (t1.x - t2.x) * (t1.x - t2.x) + (t1.y - t2.y) * (t1.y - t2.y);
}

bool jeKvadrat(const Tacka& t1, const Tacka& t2, const Tacka& t3, const Tacka& t4) {
  return kvadratRastojanja(t1, t2) == kvadratRastojanja(t2, t3) &&
    kvadratRastojanja(t2, t3) == kvadratRastojanja(t3, t4) &&
    kvadratRastojanja(t3, t4) == kvadratRastojanja(t4, t1) &&
    kvadratRastojanja(t1, t3) == kvadratRastojanja(t2, t4);
```

```
}

int main() {
  int n;
  cin >> n;
  vector<Tacka> tacke(n);
  for (int i = 0; i < n; i++)
    cin >> tacke[i].x >> tacke[i].y;
  int brojKvadrata = 0;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k++)
        for (int l = 0; l < n; l++)
          if (i != j && i != k && i != l && j != k && j != l && k != l &&
              jeKvadrat(tacke[i], tacke[j], tacke[k], tacke[l]))
            brojKvadrata++;

  cout << brojKvadrata / 8 << endl;
}
```

Бинарна претрага

Квадрат над дијагоналом

Бољи начин је да се за сваки пар тачака провери да ли чини дијагоналу квадрата састављеног од тачака из тог скупа. Ако два наспрамна темена квадрата имају координате (x_1, y_1) и (x_2, y_2), вектор који их спаја има координате (d_x, d_y) где је $d_x = x_2 - x_1$ и $d_y = y_2 - y_1$. До два преостала темена квадрата може стићи тако што се почетно теме прво транслира до средишта квадрата, транслацијом за вектор $(d_x/2, d_y/2)$, а затим се транслира за вектор који је ротација тог вектора за 90 степени (ако ротирамо у једном смеру добијамо једно, а ако ротирамо у другом смеру добијамо друго теме). Ротације тог вектора су $(-d_y/2, d_x/2)$ и $(d_y/2, -d_x/2)$.

Ако добијена темена квадрата нису целобројна, можемо их одмах елиминисати. У супротном, потребно је да проверимо да ли се налазе у полазном скупу тачака. Наивни начин је да се сваки пут изврши линеарна претрага. Много боље је да се тачке сортирају некако (на пример, у лексикографском редоследу координата, тј. по координати x тј. y ако су x координате једнаке) и да се примени бинарна претрага.

Пошто ће сваки квадрат бити пронађен два пута (по једном за сваку своју дијагоналу) добијени број квадрата треба поделити са два.

Сложеност почетног сортирања је $O(n \log n)$, након чега следи $\binom{n}{2} = \frac{n(n-1)}{2}$ израчунавања других темена квадрата (што је операција константе сложености) и исто толико бинарних претрага (што је операције сложености $O(\log n)$), тако да је сложеност овог приступа $O(n^2 \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

typedef pair<int, int> Tacka;

Tacka transliraj(const Tacka& t, int dx, int dy) {
  return make_pair(t.first + dx, t.second + dy);
}

bool DrugaDvaTemenaKvadrata(const Tacka& t1, const Tacka& t2,
                            Tacka& t3, Tacka& t4) {
  int x1 = t1.first, y1 = t1.second, x2 = t2.first, y2 = t2.second;
  int dx = x2 - x1, dy = y2 - y1;
  if ((dx + dy) % 2 != 0)
    return false;
  t3 = transliraj(t1, (dx - dy) / 2, (dy + dx) / 2);
  t4 = transliraj(t1, (dx + dy) / 2, (dy - dx) / 2);
```

```
  return true;
}

int main() {
  int n;
  cin >> n;
  vector<Tacka> tacke(n);
  for (int i = 0; i < n; i++)
    cin >> tacke[i].first >> tacke[i].second;

  sort(tacke.begin(), tacke.end());

  int brojKvadrata = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++) {
      Tacka t3, t4;
      if (DrugaDvaTemenaKvadrata(tacke[i], tacke[j], t3, t4)) {
        if (binary_search(tacke.begin(), tacke.end(), t3) &&
            binary_search(tacke.begin(), tacke.end(), t4))
          brojKvadrata++;
      }
    }

  cout << brojKvadrata / 2 << endl;

  return 0;
}
```

Квадрат над страницом

Уместо разматрања дијагонала, могуће је проверити и да ли сваки пар тачака чини страницу квадрата, али тада би се сваки квадрат пронашао 4 пута.

И сложеност овог приступа је $O(n^2 \log n)$, једино што је број бинарних претрага двоструко повећан.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

typedef pair<int, int> Tacka;

Tacka transliraj(const Tacka& t, int dx, int dy) {
  return make_pair(t.first + dx, t.second + dy);
}

void DrugaDvaTemenaKvadrata(const Tacka& t1, const Tacka& t2,
                            Tacka& t3_1, Tacka& t4_1,
                            Tacka& t3_2, Tacka& t4_2) {
  int dx = t2.first - t1.first, dy = t2.second - t1.second;
  t3_1 = transliraj(t1, -dy, dx);
  t4_1 = transliraj(t2, -dy, dx);
  t3_2 = transliraj(t1, dy, -dx);
  t4_2 = transliraj(t2, dy, -dx);
}

int main() {
  int n;
  cin >> n;
  vector<Tacka> tacke(n);
```

```
  for (int i = 0; i < n; i++)
    cin >> tacke[i].first >> tacke[i].second;

  sort(tacke.begin(), tacke.end());

  int brojKvadrata = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++) {
      Tacka t3_1, t4_1, t3_2, t4_2;
      DrugaDvaTemenaKvadrata(tacke[i], tacke[j], t3_1, t4_1, t3_2, t4_2);
      if (binary_search(tacke.begin(), tacke.end(), t3_1) &&
          binary_search(tacke.begin(), tacke.end(), t4_1))
        brojKvadrata++;

      if (binary_search(tacke.begin(), tacke.end(), t3_2) &&
          binary_search(tacke.begin(), tacke.end(), t4_2))
        brojKvadrata++;
    }

  cout << brojKvadrata / 4 << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Ранг сваког елемента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: i-ти на месту i

Напиши програм који проверава да ли у строго растућем низу елемената постоји позиција i таква да се на позицији i налази вредност i тј. да важи да је $a_i = i$ (позиције се броје од нуле).

Улаз: Са стандардног улаза се уноси број n ($0 \leq n \leq 10^5$), а затим и строго растући низ од n целих бројева (сваки у посебном реду).

Излаз: На стандардни излаз исписати индекс i такав да је $a_i = i$ или текст `nema` ако такав индекс не постоји у низу. Ако у низу постоји више таквих индекса исписати најмањи од њих.

Пример

Улаз	*Излаз*
6 -3 -1 1 3 5 7	3

Решење

Линеарна претрага

Директан начин да се задатак реши је да се употреби линеарна претрага и да се позиције проверавају редом, од 0 до $n - 1$ све док се не пронађе прва позиција која задовољава услов или док се не дође до краја низа. Разни начини имплементације линеарне претраге описани су у задатку Негативан број.

Сложеност линеарне претраге је $O(n)$.

```
#include <iostream>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  int i;
  for (i = 0; i < n; i++) {
    int x;
    cin >> x;
    if (x == i) {
      cout << i << endl;
      break;
    }
  }

  if (i >= n)
    cout << "nema" << endl;

  return 0;
}
```

Бинарна претрага трансформисаног низа

Размотримо низ `-10 -4 1 3 4 9 11`. Елемент `-10` је мањи од своје позиције 0 за 10. Елемент `-4` је мањи од своје позиције `1` за `5`, елемент `1` је мањи од своје позиције 2 за 1. Елементи `3` и `4` су једнаки својим позицијама. Елемент `9` је већи од своје позиције `5` за `4` док је елемент `11` већи од своје позиције `6` за `5`. Примећујемо одређену монотоност у овом низу, што није случајно. Заиста, ако је $a_i = i$, тада је $a_i - i = 0$. Покажимо да је низ $a_i - i$ неопадајући. Посматарајмо два елемента a_i и a_j на позицијама на којима је $0 \leq i < j$. Пошто је низ a строго растући, важи да је $a_{i+1} > a_i$, па је $a_{i+1} \geq a_i + 1$. Слично је $a_{i+2} > a_{i+1}$, па је $a_{i+2} \geq a_{i+1} + 1 \geq a_i + 2$. Настављањем овог резона важи да је $a_j \geq a_i + (j - i)$. Зато је $a_j - j \geq a_i - i$. Решење, дакле, можемо одредити тако што бинарном претрагом проверимо да ли неопадајући низ $a_i - i$ садржи нулу и ако садржи, тада је решење позиција на којој се та нула налази.

Један од најлакших начина да реализујемо бинарну претрагу је да употребимо библиотечку функцију. Пошто нам је потребна прва позиција нуле у трансформисаном низу, не можемо употребити функцију `binary_search`, већ морамо употребити функцију `lower_bound`. Такав поступак је описан и у задатку Ранг сваког елемента.

Сложеност бинарне претраге је $O(\log n)$, међутим, временом доминира учитавање и трансформисање низа које захтева $O(n)$ корака.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // pripremamo ga za pretragu a[k] = k akko je a[k] - k = 0 tako da
```

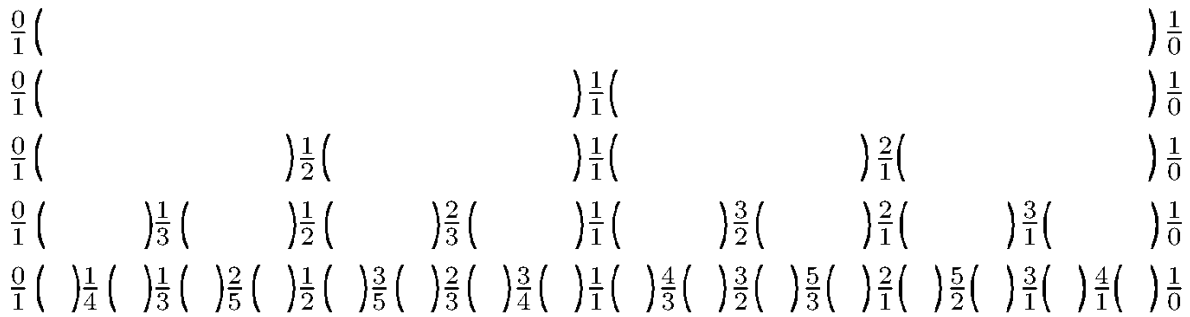
```
  // umesto niza a, pretrazujemo niz a[i] - i (koji je neopadajuci)
  for (int i = 0; i < n; i++)
    a[i] -= i;

  // trazimo poziciju nule u transformisanom nizu tako sto pronalazimo
  // poziciju prvog elementa koji je >= 0
  auto it = lower_bound(a.begin(), a.end(), 0);

  // ako takav element postoji i ako je jednak nuli
  if (it != a.end() && *it == 0)
    // pronasli smo element i izracunavamo njegovo rastojanje od pocetka niza
    cout << distance(a.begin(), it) << endl;
  else
    // u suprotnom element ne postoji u nizu
    cout << "nema" << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Штерн-Брокооово стабло

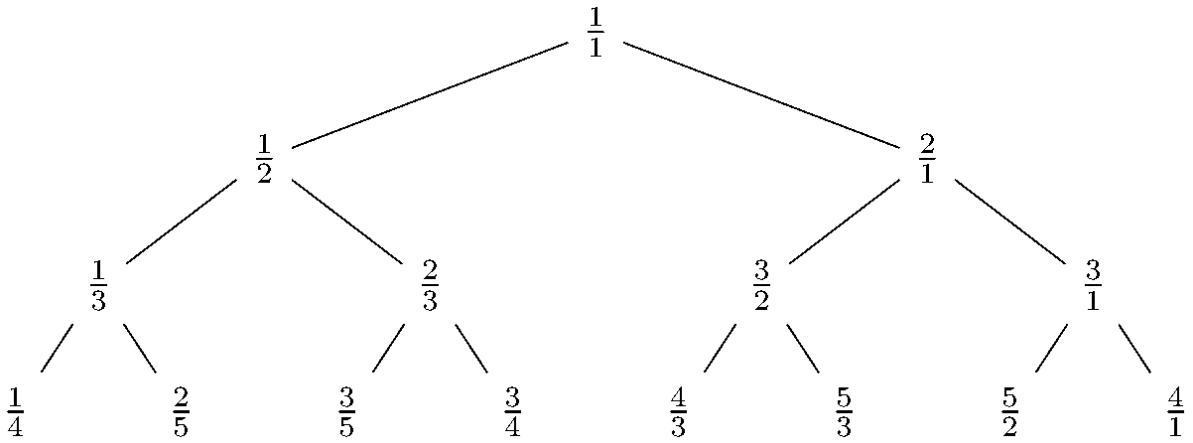
Штерн-Брокооово стабло садржи сваки позитиван, потпуно скраћен разломак тачно једном. Описаћемо један поступак којим се може добити ово стабло. Уочимо најпре да ако за природне бројеве a, b, c, d важи $\frac{a}{b} < \frac{c}{d}$, онда важи и $\frac{a}{b} < \frac{a+c}{b+d} < \frac{c}{d}$. Одавде следи да интервал $(\frac{a}{b}, \frac{c}{d})$ може да се тачком $\frac{a+c}{b+d}$ подели да два интервала, $(\frac{a}{b}, \frac{a+c}{b+d})$ и $(\frac{a+c}{b+d}, \frac{c}{d})$.

Узмимо сада за полазни интервал $(\frac{0}{1}, \frac{1}{0})$ тј. $(0, +\infty)$. У првом кораку га на описани начин поделимо на два интервала, а у сваком следећем кораку сваки од интервала које имамо делимо даље на два подинтервала. Тако добијамо следећу поделу позитивног дела x осе на интервале.



Слика 2.6: Подела интервала

Штерн-Брокооово стабло се састоји управо од деоних тачака ове поделе (тачке добијене у -том кораку поделе представљају чворове -тог нивоа).



Слика 2.7: Штерн-Брокоово стабло

Тако се на пример, у корену налази разломак $\frac{1}{1}$, испод њега се налазе $\frac{1}{2}$ и $\frac{2}{1}$, на наредном нивоу се налазе $\frac{1}{3}$, $\frac{2}{3}$, $\frac{3}{2}$ и $\frac{3}{1}$ итд. При томе се до сваког позитивног разломка може стићи од корена на јединствен начин, крећући се у сваком кораку низ леву или десну грану. На пример, до разломка $\frac{3}{5}$, путања је `LDL` (лево, десно, лево).

Напиши програм који за унети разломак одређује путању којом се долази до њега.

Улаз: Са стандардног улаза се учитавају позитивни, узајамно прости бројеви m и n ($m, n \leq 10^6$), раздвојени једним размаком.

Излаз: На стандардни излаз исписати путању до разломка $\frac{m}{n}$.

Пример 1

Улаз	*Излаз*
`3 5`	`LDL`

Пример 2

Улаз	*Излаз*
`1155 8398`	`LLLLLLLDDDLDDLLLLDDLLDDD`

Решење

Захваљујући томе што је низ разломака на сваком нивоу стабла сортиран по величини, Задатак се може решити бинарном претрагом. У сваком кораку ћемо одржавати леву и десну границу интервала коме задати разломак припада. Леву границу $\frac{a_l}{b_l}$ иницијализујемо на $\frac{0}{1}$ тј. 0, а десну границу $\frac{a_d}{b_d}$ иницијализујемо на $\frac{1}{0}$ тј. $+\infty$. У сваком кораку налазимо "средину" интервала тако што израчунамо $\frac{a}{b} = \frac{a_l+a_d}{b_l+b_d}$. Ако је наш разломак $\frac{m}{n}$ једнак добијеном разлому, поступак се завршава, ако је мањи од њега мењамо десну границу постављајући $\frac{a_d}{b_d} = \frac{a}{b}$, а ако је већи од њега мењамо леву границу постављајући $\frac{a_l}{b_l} = \frac{a}{b}$. Пошто дрво садржи све разломке, поступак ће се сигурно завршити у коначном броју корака.

```
#include <iostream>
using namespace std;

int main() {
  long long m, n;
  cin >> m >> n;

  long long a1 = 0, b1 = 1;
  long long a2 = 1, b2 = 0;

  while (true) {
    long long a = a1 + a2;
    long long b = b1 + b2;
    if (m*b < a*n) {
      cout << 'L';
      a2 = a; b2 = b;
    } else if (m*b > a*n) {
      cout << 'D';
      a1 = a; b1 = b;
```

```
    } else
      break;
  }
  cout << endl;

  return 0;
}
```

2.9.2 Бинарна претрага преломне тачке

У свом најопштијем облику, бинарна претрага се може формулисати на следећи начин. Размотримо низ елемената такав да су елементи подељени у две групе, на основу неког својства P. Елементи у почетном делу низа су сви такви да немају то својство P, а елементи у завршном делу низа су сви такви да имају својство P. Низ је, дакле, облика `----++++++++`, где су са `-` означени елементи који немају, а са `+` елементи који имају својство P. Могућа је и ситуација у којој је нека од група празна. Својство P може бити сасвим произвољно. На пример, у сортираном низу бројева можемо посматрати својство *већи је или једнак* X за неку дату вредност X. Тада се у првом делу низа налазе елементи који нису већи или једнаки X, тј. строго су мањи од X, док се иза њих налазе елементи који имају својство, тј. већи су или једнаки X. Даље, на пример, низ ученика може бити организован тако да су прво наведени дечаци, а затим девојчице.

Бинарна претрага нам може помоћи да ефикасно одредимо *преломну тачку*, тј. место где престаје једна и почиње друга група елемената. То може бити или позиција последњег елемента који нема својство P или првог елемента који има својство P. Ако сви елементи низа имају својство P, тада претрага за позицијом последњег елемента низа који нема својство треба да врати -1. Ако ниједан елемент низа нема својство P, тада претрага за позицијом првог елемента низа који има својство P треба да врати дужину низа. Познавање преломне тачке нам омогућава и да ефикасно одговоримо на питање колико је елемената у свакој групи (колико елемената низа нема, а колико елемената низа има својство P).

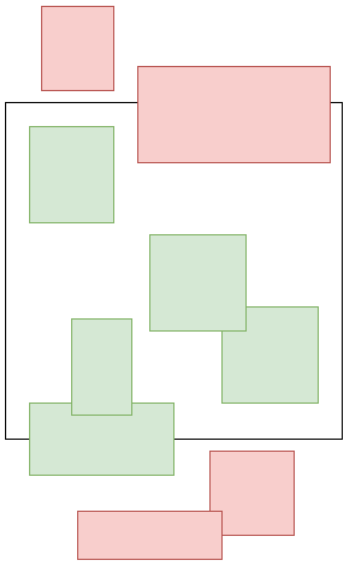
Класична бинарна претрага се лако формулише као претрага преломне тачке. Ако у низу пронађемо позицију првог елемента који је већи или једнак траженој вредности X, тада можемо проверити да ли је та позиција унутар низа (строго мања од дужине низа) и да ли се на њој налази елемент X - ако је то испуњено елемент постоји у низу, а у супротном не постоји.

У наставку ће кроз низ задатака бити приказан алгоритам бинарне претраге преломне тачке у низу. У првим примерима у низу бројева ћемо тражити позицију првог елемента који је (строго или нестрого) већи или мањи од дате вредности, док ћемо у наредним примерима посматрати и другачије низове, подељене у односу на неко својство P.

Позиција преломне тачке може бити пронађена и уз помоћ инвентивне употребе функција `lower_bound` и `upper_bound`.

Задатак: Први већи и последњи мањи

Веб-страница садржи неколико објеката правоугаоног облика (слика, пасуса, табела и слично). За сваки објекат је познат положај његове горње и доње ивице у односу на врх странице. Приликом приказа странице и померања приказа (скроловања) многи објекти се не приказују. Систем ради тако што приказује све оне објекте чија се горња ивица налази унутар приказаног дела странице (како је приказано на слици). Напиши програм који одређује објекте чија се горња ивица види током приказа одређених делова странице.



Слика 2.8: Приказани и скривени објекти

Улаз: Са стандардног улаза учитава се број n ($1 \leq n \leq 50000$), а затим n линија које садрже парове природних бројева мањих од 10^6 раздвојене са по једним размаком, а који представљају положај (удаљеност од врха странице) горње и доње ивице n објеката са странице. Објекти су поређани неопадајуће у односу на положај њихове горње ивице.

Након тога се уноси број m ($1 \leq m \leq 50000$), а затим m парова природних бројева раздвојених са по једним размаком (њих највише 50000) који представљају положај (удаљеност од врха странице) горње и доње границе видљивог дела странице.

Излаз: За сваки пар бројева који одређују видљиви део странице на стандардни излаз исписати по једну линију која садржи два цела броја раздвојена размаком. Први број представља позицију (индекс унутар низа, бројано од нуле) првог објекта чији је положај горње ивице строго већи од положаја горње границе видљивог дела странице, а други број представља позицију последњег објекта чији је положај горње ивице строго мањи од положаја доње границе видљивог дела странице. Ако се горње ивице свих објеката у низу налазе изнад горње границе видљивог дела, први број треба да буде једнак n. Ако се горње ивице свих објеката у низу налазе испод доње границе, други број треба да буде -1.

Пример

Улаз	*Излаз*
11	3 8
3 5	3 8
4 7	5 9
5 7	0 10
8 13	11 10
8 16	0 -1
9 11	
11 17	
11 12	
11 20	
13 20	
14 18	
6	
5 12	
6 13	
8 14	
0 20	
20 25	
1 2	

Објашњење

У првом упиту видљив део странице је од 5 до 12, први видљив објекат је $(8, 13)$ на позицији 3 у низу (код објекта $(5, 7)$ горња ивица није строго већа од врха странице 5, као што се тражи), а последњи видљив је $(11, 20)$ на позицији 8 у низу. Слично важи и за наредна три упита.

У упиту $(20, 25)$, сви врхови почињу изнад висине 20, па први објекат који почиње изнад границе 20 не постоји (зато се исписује вредност 11), док је објекат (14, 18), на позицији 10 последњи који почиње испод границе 20.

У упиту $(1, 2)$ сви врхови почињу испод висине 2, па је објекат (3, 5) на позицији 0 први који почиње испод висине 1, док не постоји ни један објекат који почиње изнад висине 2 (па ни последњи такав) и као други број исписује се -1.

Решење

По условима задатка релевантне су само висине горњих ивица објеката, тако да ћемо само њих учитати у низ, док ћемо доње ивице занемаривати. Потребно је имплементирати функције које проналазе позицију првог елемента у низу чија је вредност строго већа од датог броја и позицију последњег елемента у низу чија је вредност строго мања од датог броја.

Линеарна претрага

Наиван начин да пронађемо позицију првог елемента у низу чија је вредност строго већа од дате је да извршимо линеарну претрагу тј. да проверавамо један по један елемент низа a све док не наиђемо на први који је већи од датог елемента x или док не дођемо до краја низа a (што ће се десити у случају када су сви елементи у низу a мањи или једнаки x). Проналазак последњег елемента у низу чија је вредност строго мања од датог броја се може имплементирати тако што се пронађе први елемент у низу чија је вредност већа или једнака датом броју, а затим се као резултат узме позиција за један мања од те.

Сложеност овог приступа је линеарна (и износи $O(n)$), што значи да се за m различитих позиција скрола користи квадратни број операција $O(mn)$. Приметимо да другу претрагу можемо започети од позиције која је пронађена у првој, јер је висина (удаљеност од почетка странице) доњег дела скрола већа од висине његовог горњег дела, међутим, ово неће побољшати асимптотску сложеност.

Пошто се у овом задатку преплићу фаза учитавања и исписа података, потребно је оптимизовати улаз и излаз и прилагодити га аутоматском тестирању (помоћу `cin.tie(0)` и коришћења `\n` уместо `endl`).

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;
```

```
int prviVeci(const vector<int>& a, int x) {
  for (int i = 0; i < a.size(); i++)
    if (a[i] > x)
      return i;
  return a.size();
}

int poslednjiManji(const vector<int>& a, int x) {
  for (int i = 0; i < a.size(); i++)
    if (a[i] >= x)
      return i-1;
  return a.size() - 1;
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  // ucitavamo niz gornjih ivica objekata
  int n;
  cin >> n;
  vector<int> gornjeIvice(n);
  for (int i = 0; i < n; i++) {
    cin >> gornjeIvice[i];
    // donje ivice su irelevantne, pa ih ne pamtimo
    int _;
    cin >> _;
  }

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    // ucitavamo gornju i donju granicu
    int gornjaGranica, donjaGranica;
    cin >> gornjaGranica >> donjaGranica;
    // pozicija prvog objekta ciji je vrh strogo veci od pocetka skrola
    int pv = prviVeci(gornjeIvice, gornjaGranica);
    int pm = poslednjiManji(gornjeIvice, donjaGranica);
    cout << pv << " " << pm << '\n';
  }

  return 0;
}
```

Бинарна претрага

Претходно неефикасно решење не узима у обзир чињеницу да је низ горњих граница уређен неопадајуће. Неупоредиво ефикаснији приступ заснован је на бинарној претрази, коју смо видели у задатку Провера бар-кодова. Прикажимо како можемо пронаћи позицију првог елемента у низу који је строго већи од неког датог.

Уведимо променљиве l и d и наметнимо услов (инваријанту) да све време током претраге важи да су:

- елементи низа a на позицијама из интервала $[0, l)$ мањи или једнаки вредности x,
- елементи на позицијама из интервала $[d, n)$ строго већи од x.

Елементима на позицијама из интервала $[l, d)$ статус још није познат. Овај услов ће бити иницијализован ако се променљива l иницијализује на нулу, а променљива d на вредност n.

Нека s представља средину интервала $[l, d)$.

- Ако је елемент низа a на позицији s мањи или једнак вредности x такви су и сви елементи из интервала $[l, s]$. Зато можемо вредност l поставити на $s + 1$.

- Ако је елемент низа a на позицији s већи од вредности x тада су и сви елементи интервала $[s, d)$ строго већи од x. Зато можемо вредност d поставити на s.

Претрагу вршимо све док постоје елементи непознатог статуса, тј. док је интервал $[l, d)$ непразан, односно док је $l < d$. У тренутку када је $l = d$, на основу инваријанте знамо да се први елемент строго већи од x налази на позицији d (јер су сви елементи у интервалу $[0, l)$ тј. $[0, d)$ мањи или једнаки x, док су елементи на позицијама $[d, n)$ строго већи од x и први такав елемент се налази на позицији d (осим у случају када је $d = n$, када су сви елементи низа a мањи или једнаки x, при чему је и у том случају $d = n$ управо тражена вредност).

Напоменимо да смо програм могли засновати и на инваријанти да све време током претраге важи да су:

- елементи низа a на позицијама из интервала $[0, l)$ мањи или једнаки од x,
- елементи низа a на позицијама из интервала (d, n) строго већи од x.

Тада се у интервалу $[l, d]$ налазе елементи чији статус још није познат (такво решење описано је у задатку Провера бар-кодова). Тада бисмо l иницијализовали на 0, а d на $n - 1$. Претрага се врши док је $l \leq d$. Након проналажења средине s интервала $[l, d]$, ако је $a[s] > x$, тада се d може поставити на $s - 1$ (јер су сви елементи од позиције s па навише строго већи од x), а у супротном се l може поставити на $s + 1$ (јер су сви елементи на позицијама лево од s закључно са њом мањи или једнаки x).

Позицију последњег елемента који је строго мањи можемо најједноставније пронаћи поново тако што претходни приступ модификујемо да врати прву позицију елемента који је већи или једнак датој вредности (потребно је променити само један оператор поређења у претходном коду), а затим вратимо претходну позицију у низу.

Сложеност сваке бинарне претраге је $O(\log n)$ па се m упита извршава у времену $O(m \log n)$.

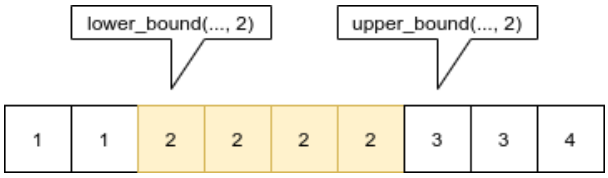
Пошто се у овом задатку преплићу фаза учитавања и исписа података, потребно је оптимизовати улаз и излаз и прилагодити га аутоматском тестирању (помоћу `cin.tie(0)` и коришћења `\n` уместо `endl`).

```
#include <iostream>
#include <vector>

using namespace std;

// u nizu a pronalazi i vraca poziciju prvog elementa koji je veci od x,
// tj. n ako takvog elementa u nizu nema
int prviVeci(const vector<int>& a, int n, int x) {
  // [0, l) - elementi manji ili jednaki od x
  // [l, d) - nepoznati
  // [d, n) - elementi strogo veci od x

  int l = 0, d = n;
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] > x)
      d = s;
    else
      l = s + 1;
  }
  // prvi element strogo veci od x se nalazi na poziciji l=d
  return d;
}

// u nizu a pronalazi i vraca poziciju poslednjeg elementa
// koji je strogo manji od x, tj. -1 ako takvog elementa u nizu
// nema
int poslednjiManji(const vector<int>& a, int n, int x) {
  // pronalazimo poziciju prvog elementa koji je veci ili jednak x

  // [0, l) - elementi strogo manji od x
```

```
  // [l, d) - nepoznati
  // [d, n) - veci ili jednaki od x
  int l = 0, d = n;
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] >= x)
      d = s;
    else
      l = s + 1;
  }

  // prvi element veci ili jednak od x se nalazi na poziciji l=d
  // poslednji strogo manji od x se nalazi na prethodnoj poziciji
  return d - 1;
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  // ucitavamo niz gornjih ivica objekata
  int n;
  cin >> n;
  vector<int> gornjeIvice(n);
  for (int i = 0; i < n; i++) {
    cin >> gornjeIvice[i];
    // donje ivice su irelevantne, pa ih preskacemo
    int _;
    cin >> _;
  }

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    // ucitavamo gornju i donju granicu
    int gornjaGranica, donjaGranica;
    cin >> gornjaGranica >> donjaGranica;
    // pozicija prvog objekta ciji je vrh strogo veci od pocetka skrola
    int pv = prviVeci(gornjeIvice, n, gornjaGranica);
    int pm = poslednjiManji(gornjeIvice, n, donjaGranica);
    cout << pv << " " << pm << "\n";
  }

  return 0;
}
```

Библиотечке функције

У језику C++ на располагању нам је библиотечка функција `upper_bound` која враћа итератор који указује на први елемент низа који је строго већи од траженог (ако су сви елементи мањи или једнаки од траженог, функција враћа итератор који указује непосредно иза краја низа). Функцијом `lower_bound` која је веома слична функцији `upper_bound` добијамо итератор који указује на први елемент који је већи или једнак од траженог (ако су сви елементи мањи од траженог, функција враћа итератор који указује непосредно иза краја низа).



Слика 2.9: Функције lower i upper bound

Обе функције примају итератор на почетак низа који се претражује, итератор који указује непосредно иза краја низа и елемент који се тражи. Функцијом `distance` можемо израчунати растојање између добијеног итератора и почетка низа и тако добити позицију у низу првог елемента који је строго већи тј. већи или једнак од траженог. Сложеност обе функције је логаритамска у односу на број елемената у распону итератора који се претражује.

Позиција последњег елемента који је строго мањи од траженог је једна позиција испред првог елемента који је већи или једнак од траженог (а који можемо одредити функцијом `lower_bound`). Позиција последњег елемента који је мањи или једнак од траженог је једна позиција испред првог елемента који је већи или једнак од траженог (а који можемо одредити функцијом `upper_bound`).

Сложеност обе библиотечке функције је $O(\log n)$. Пошто се претрага понавља m пута, укупна сложеност је $O(m \log n)$.

Пошто се у овом задатку преплићу фаза учитавања и исписа података, потребно је оптимизовати улаз и излаз и прилагодити га аутоматском тестирању (помоћу `cin.tie(0)` и коришћења `\n` уместо `endl`).

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

using namespace std;

int prviVeciIliJednak(const vector<int>& a, int x) {
  return distance(a.begin(), upper_bound(a.begin(), a.end(), x));
}

int poslednjiManji(const vector<int>& a, int x) {
  return distance(a.begin(), lower_bound(a.begin(), a.end(), x)) - 1;
}


int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  // ucitavamo niz gornjih ivica objekata
  int n;
  cin >> n;
  vector<int> gornjeIvice(n);
  for (int i = 0; i < n; i++) {
    cin >> gornjeIvice[i];
    // donje ivice su irelevantne, pa ih preskacemo
    int _;
    cin >> _;
  }

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    // ucitavamo gornju i donju granicu
    int gornjaGranica, donjaGranica;
```

```
    cin >> gornjaGranica >> donjaGranica;
    // pozicija prvog objekta ciji je vrh strogo veci od pocetka skrola
    int pv = prviVeciIliJednak(gornjeIvice, gornjaGranica);
    int pm = poslednjiManji(gornjeIvice, donjaGranica);
    cout << pv << " " << pm << "\n";
  }

  return 0;
}
```

Задатак: Провера бар-кодова

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Бинарна претрага првог елемента који је већи или једнак од траженог

Један начин да се реализује бинарна претрага којом се проверава да ли елемент постоји у низу је да се пронађе позиција првог елемента који је већи или једнак траженом елементу x. Број x се јавља у низу ако и само у низу постоји елемент који је већи или једнак x и први такав елемент је управо x.

Овај приступ је описан у задацима Ранг сваког елемента и i-ти на месту i, у којима је описана употреба функције `lower_bound` која проналази позицију првог елемента који је већи или једнак од датог.

Опишимо имплментацију без употребе библиотечких функција. Први елемент који је већи или једнак од елемента x можемо наћи на следећи начин. Сличан поступак је приказан у задатку Први већи и последњи мањи.

Уведимо промењљиве l и d и наметнимо услов (инваријанту) да све време током претраге важи $0 \leq l \leq d + 1 \leq n$, и да су

- елементи низа a на позицијама из интервала $[0, l)$ мањи од x,
- елементи на позицијама из интервала (d, n) већи или једнаки x.

Елементима на позицијама из интервала $[l, d]$ статус још није познат. Овај услов ће бити иницијализован ако се промењљива l иницијализује на нулу, а промењљива d на вредност $n - 1$.

Нека s представља средину интервала $[l, d]$. Важи $l \leq s \leq d$.

- Ако је елемент низа a на позицији s већи или једнак вредности x тада су, услед сортираности низа у неопадајућем поретку, и сви елементи интервала $[s, n)$ су већи или једнаки x. Зато можемо вредност d поставити на $s - 1$ и инваријанта ће остати да важи. Заиста, важи да је $d' = s - 1$, па су елементи из интервала $(d', n) = (s - 1, n) = [s, n)$ већи или једнаки x, док су сви елементи из интервала $[0, l') = [0, l)$ строго мањи од x, што знамо на основу претпоставке. Услов $0 \leq l' \leq d' + 1 \leq n$, еквивалентан је услову $0 \leq l \leq s \leq n$, што сигурно важи, јер је $l \leq s \leq d$ и $0 \leq l \leq d + 1 \leq n$.
- Ако је елемент низа a на позицији s строго мањи од вредности x такви су, услед сортираности низа у неопадајућем поретку, и сви елементи из интервала $[0, s]$. Зато можемо вредност l поставити на $s + 1$ и инваријанта ће остати да важи. Заиста, важи да је $l' = s + 1$, па су сви елементи из интервала $[0, l') = [0, s + 1) = [0, s]$ строго мањи од x, док су сви елементи из интервала $(d', n) = (d, n)$ већи или једнаки од x, што знамо на основу претпоставке. Услов $0 \leq l' \leq d' + 1 \leq n$, еквивалентан је услову $0 \leq s + 1 \leq d + 1 \leq n$, што сигурно важи, јер је $l \leq s \leq d$ и $0 \leq l \leq d + 1 \leq n$.

Претрагу вршимо све док постоје елементи непознатог статус, тј. док је интервал $[l, d]$ непразан, односно док је $l \leq d$. У тренутку када је $l > d$, када се претрага заврши, важи да је $l = d + 1$. На основу инваријанте знамо да се први елемент већи или једнак x налази на позицији $l = d + 1$, јер су сви елементи у интервалу $(d, n) = [d+1, n)$ већи или једнаки од x. Изузетак је случај када је $l = n$, када су сви елементи низа a строго мањи од x.

Тражени елемент x постоји у низу ако и само ако је $l = d + 1 < n$ и ако је $a_l = x$.

Алгоритам се сигурно зауставља јер се у сваком кораку ширина интервала $[l, d]$ која је једнака $d-l+1$ строго смањује, док не достигне нулу.

Приметимо да овим алгоритмом пролазимо позицију првог појављивања елемента x у низу, док алгоритмом који проверава једнакост током претраге проналазимо било коју позицију. Пошто је често потребно наћи баш прво појављивање елемента, овај алгоритам је јасно у предности.

Пошто се у сваком кораку интервал $[l, d]$ полови, пошто иницијално крећемо од интервала $[0, n-1]$ и пошто се претрага завршава када се интервал испразни, број потребних корака да се то догоди је $O(\log n)$. Пошто се независно претражује m елемената, укупна сложеност је $O(m \log n)$. Приметимо да за разлику од варијанте у којој се унутар тела петље проверава и услов једнакости, за шта су потребна два поређења, у овом алгоритму се врши само једно поређење у телу петље, па је константни фактор мало мањи (што је често занемариво).

```
#include <iostream>

using namespace std;

// funkcija proverava da li se u datom sortiranom nizu a duzine n
// nalazi element x
bool sadrzi(int a[], int n, int x) {
  // trazimo poziciju prvog elementa u nizu a koji je veci ili jednak x

  // [0, l) - elementi strogo manji od x
  // (d, n) - elementi veci ili jednaki x
  // [l, d] - nepoznati elementi
  int l = 0, d = n - 1;
  // dok god taj interval nije prazan
  while (l <= d) {
    // nalazimo sredinu intervala
    int s = l + (d - l) / 2;

    if (a[s] >= x)
      // posto je niz sortiran svi posle pozicije s-1 su veci ili jednaki x
      d = s - 1;
    else
      // posto je niz sortiran svi pre pozicije s+1 su strogo manji od x
      l = s + 1;
  }
  // prvi veci ili jednak nalazi se na poziciji d+1 = l

  // x je u nizu ako i samo ako u nizu postoji element koji je veci
  // ili jednak x i ako je prvi takav element upravo x
  return l < n && a[l] == x;
}

// maksimalni broj elemenata niza dat tekstom zadatka
const int MAX = 50000;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  int a[MAX];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // broj onih koji postoje u nizu
  int broj = 0;
  // ucitavamo broj po broj do kraja ulaza
  int x;
```

```
  while (cin >> x) {
    // ako je broj sadrzan u nizu, uvecavamo brojac
    if (sadrzi(a, n, x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Број такмичара изнад прага

Државна комисија треба да одреди праг за пролазак такмичара са окружног на државно такмичење. Пошто је информатика постала обавезан предмет у основним школама, број такмичара је јако велики. Администраторку Мају која одржава табелу са резултатима стално питају који би број такмичара прошао даље када би праг пролазности био толико и толико поена (даље се пласирају сви ученици чији је број поена већи или једнак прагу). Одлучила је да напише програм који даје одговор на та питања.

Улаз: Са стандардног улаза учитава се број такмичара n ($0 \leq n \leq 10^5$), а затим и поени такмичара (цели бројеви), задати у сортираном редоследу од највећег до најмањег. Након тога се учитава број m ($1 \leq m \leq 50000$) који представља број питања на која Маја треба да одговори, а затим и m бројева за које је потребно дати одговор колико би се такмичара пласирало када би се тај број узео за праг. Сваки број се налази у посебном реду.

Излаз: На стандардни излаз исписати тражене бројеве пласираних такмичара, сваки у посебном реду.

Пример

Улаз	*Излаз*
5	0
89	4
73	3
73	5
56	
23	
4	
95	
50	
70	
0	

Задатак: Тастатура и миш

Огњен жели да купи тастатуру и миш за одређену своту новца које има. Напиши програм који на основу доступних цена тастатура и мишева одређује која је највећа свота новца за коју може да се купи једна тастатура и један миш.

Улаз: Са стандардног улаза се учитава број доступних тастатура t ($1 \leq t \leq 10^5$), затим t природних бројава мањих од 10^6 раздвојених размацима који представљају цене тастататура. Након тога се се учитава број доступних мишева m ($1 \leq m \leq 10^5$) и затим m природних бројава мањих од 10^6 раздвојених размацима који представљају цене мишева. У последњем реду се налази буџет којим Огњен располаже.

Излаз: На стандардни излаз исписати тражену максималну цену једне тастатуре и једног миша које Огњен може да купи.

Пример

Улаз	*Излаз*
2	9
3 1	
3	
5 8 2	
10	

Решење

Груба сила

Решење грубом силом се заснива на томе да анализира сваки пар тастатуре и миша и да се пронађе максимални збир пара који је мањи или једнак датом буџету.

Сложеност овог решења је $O(n \cdot m)$, где је n број тастатура, а m број мишева.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int broj_tastatura;
  cin >> broj_tastatura;
  vector<int> tastature(broj_tastatura);
  for (int i = 0; i < broj_tastatura; i++)
    cin >> tastature[i];

  int broj_miseva;
  cin >> broj_miseva;
  vector<int> misevi(broj_miseva);
  for (int i = 0; i < broj_miseva; i++)
    cin >> misevi[i];

  int budzet;
  cin >> budzet;

  int maks_cena = -1;
  for (int t : tastature)
    for (int m : misevi)
      if (t + m <= budzet)
        maks_cena = max(maks_cena, t + m);

  cout << maks_cena << endl;

  return 0;
}
```

Бинарна претрага

Ако соритрамо низ цена мишева, тада за сваку цену тастатуре можемо бинарном претрагом пронаћи највећу цену миша који можемо купити заједно са том тастатуром. Тражимо последњу вредност која је мања или једнака од неке границе (она се налази испред прве вредности строго већа од те границе). У имплементацији је најједноставније употребити библиотечку функцију `upper_bound`.

Учитавање је сложености $O(m + n)$, сортирање цена мишева је сложености $O(n \log n)$, док се бинарна претрага затим понавља m пута и сложености је $\log n$, па је укупна сложеност $O(m + n) \log n$. Да смо сортирали цене тастатура, сложеност би била $O(m + n) \log m$, што није превише различито. Сортирањем дужег низа скраћује се фаза претраге, међутим, само сортирање односи више времена.

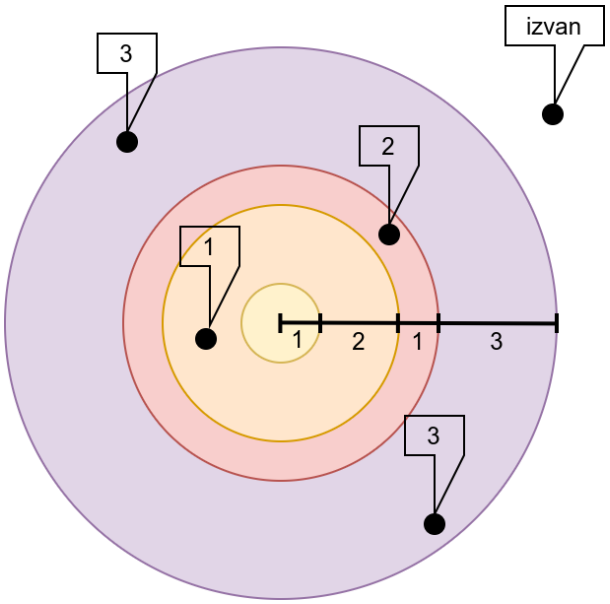
```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int broj_tastatura;
```

```
  cin >> broj_tastatura;
  vector<int> tastature(broj_tastatura);
  for (int i = 0; i < broj_tastatura; i++)
    cin >> tastature[i];

  int broj_miseva;
  cin >> broj_miseva;
  vector<int> misevi(broj_miseva);
  for (int i = 0; i < broj_miseva; i++)
    cin >> misevi[i];

  int budzet;
  cin >> budzet;

  // sortiramo niz cena miseva, da bi bilo moguce primenjivati
  // binarnu pretragu
  sort(begin(misevi), end(misevi));
  // trazena maksimalna cena
  int maks_cena = -1;
        // analiziramo sve tastature
  for (int t : tastature) {
    // odredjujemo najskuplji mis koji moze da se kupi u kombinaciji
    // sa trenutnom tastaturom
    auto it = upper_bound(begin(misevi), end(misevi), budzet - t);
    // ako takav ne postoji, prekidamo analizu trenutne tastature
    if (it == begin(misevi))
      continue;
    // azuiramo maksimalnu cenu ako je potrebno
    int cena = t + *prev(it);
    maks_cena = max(maks_cena, cena);
  }

  cout << maks_cena << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Кружне зоне

Квалитет сигнала зависи од удаљености тачке од предајника. Простор је подељен у зоне облика кружних прстенова, при чему ширине прстенова могу бити међусобно различите (како је приказано на слици). Напиши програм који за дату тачку одређује зону којој припада.



Слика 2.10: Кружне зоне

Улаз: Са стандардног улаза уноси се број n ($1 \leq n \leq 50000$), а затим и n реалних бројева заокружених на две децимале, сваки у посебном реду, који представљају ширине свих кружних прстенова (за почетни прстен, тај број представља полупречник). Након тога се уноси број m ($1 \leq m \leq 50000$) и затим m парова координата тачака (у сваком реду се налазе два реалана броја заокружена на две децимале, раздвојена са по једним размаком).

Излаз: На стандардни излаз исписати m линија. У свакој линији исписати или индекс зоне (броје се од нуле) којој тачка припада или текст `izvan` ако је тачка изван последње зоне. Ако је тачка на граници две зоне, сматрати да припада унутрашњој.

Пример

Улаз	*Излаз*
`3`	`0`
`2.0`	`1`
`3.0`	`2`
`7.0`	`izvan`
`5`	`2`
`1.0 1.0`	
`2.0 3.0`	
`8.0 7.0`	
`13.2 14.5`	
`0.0 12.0`	

Задатак: Разлика висина

У једном одељењу бирају се глумци за школску представу “Станлио и Олио”. Ови глумци су познати по томе што им је била велика разлика у висини. Напиши програм који одређује на колико начина можемо да одаберемо два глумца из одељења тако да им је разлика једнака датом броју r.

Улаз: Са стандардног улаза се уноси прво позитиван природан број r, у наредном реду број ученика у одељењу n ($1 \leq n \leq 50000$), а након тога у наредних n редова висина сваког ученика у милиметрима.

Излаз: На стандардни излаз испиши број парова које је могуће формирати.

Пример

Улаз	*Излаз*
2350 5 15745 18095 15745 16234 13395	4

Решење

Сортирање

Као и у многим проблемима претраге, сортирање низа може довести до ефикаснијих решења. За почетак, ако је низ сортиран, довољно је само да проверавамо парове такве да је други елемент пара иза првог. Дакле, за сваки елемент низа одређујемо број елемената иза њега који са њим дају тражену разлику (он је умањилац, а тражимо потенцијалне умањенике). Пошто је низ сортиран, чим наиђемо на први елемент иза њега који има већу разлику од тражене, такви ће бити и сви елементи у наставку низа, па можемо извршити одсецање и прећи на обраду следећег елемента (умањиоца).

У најгорем случају не долази до одсецања, а проверавају се сви парови којих има $\binom{n}{2} = \frac{n(n-1)}{2}$, па је сложеност овог приступа $O(n^2)$. Наравно, у сложеност је укључена и сложеност сортирања $O(n \log n)$, међутим, она је у овом случају занемарива у односу на сложеност испитивања свих парова.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int razlika;
  cin >> razlika;
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  sort(begin(a), end(a));

  int broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      if (a[j] - a[i] == razlika)
        broj++;
      else if (a[j] - a[i] > razlika)
        break;

  cout << broj << endl;

  return 0;
}
```

Бинарна претрага

Пошто је низ сортиран, решење је брже ако се потенцијални умањиоци траже бинарном претрагом (уместо линеарном).

Да би важило $a_j - a_i = r$ потребно и довољно је да важи да је $a_j = a_i + r$. Pošto je $r > 0$, a niz je sortiran

неопадајуће, a_j се, ако постоји, налази на позицијама иза i. Дакле, за сваки елемент низа a_i у делу низа иза њега проверавамо да ли постоји елемент $a_i + r$. Пошто је низ сортиран, то можемо ефикасно извршити бинарном претрагом. Да у низу нема понављања, за сваки пронађени елемент $a_i + r$ увећавали бисмо бројач парова за 1.

Међутим, пошто може бити понављања потребно је да израчунамо број појављивања елемента $a_i + r$ и да бројач увећамо за ту вредност (јер се свако пронађено појављивање када се укомбинује са елементом a_i даје један такав пар). Ако се елемент a_i појављује више пута, тада можемо уштедети број позива бинарне претраге тако што израчунамо број појављивања елемента a_i (нека је то b_i), елемента $a_j = a_i + r$ (нека је то b_j) и увећати бројач за $b_i \cdot b_j$.

Библиотечке функције

У језику C++ број појављивања елемента у сортираном низу можемо најбоље одредити библиотечком функцијом `equal_range` која враћа пар итератора - на први елемент који није мањи од датог и на први елемент који је већи од датог (оба итератора указују непосредно иза краја низа ако описани елементи не постоје). Ова функција заправо даје пар итератора који би се добили позивима редом функција `lower_bound` и `upper_bound`, које су описане у задатку Први већи и последњи мањи. Растојање између та два итератора одређује број појављивања траженог елемента. Бројање елемената a_i можемо остварити једноставним увећавањем бројача док год је наредни елемент низа једнак претходном.

Сложеност једне бинарне претраге је гарантовано $O(\log n)$. Пошто се претрага у најгорем случају врши за сваки елемент низа, укупна сложеност је $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int razlika;
  cin >> razlika;
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  sort(begin(a), end(a));

  int broj_parova = 0;
  int broj_pojavljivanja_ai = 1;
  for (int i = 1; i < n; i++) {
    if (a[i] == a[i-1])
      broj_pojavljivanja_ai++;
    else {
      auto range = equal_range(next(begin(a), i), end(a), a[i-1] + razlika);
      int broj_pojavljivanja_aj = distance(range.first, range.second);
      broj_parova += broj_pojavljivanja_ai * broj_pojavljivanja_aj;
      broj_pojavljivanja_ai = 1;
    }
  }
  cout << broj_parova << endl;
  return 0;
}
```

Ручна имплементација

Уместо библиотечке функције, можемо сами имплементирати функцију која врши бинарну претрагу и то тако што направимо варијанту која налази позицију првог елемента који је већи или једнак од датог и варијанту

која налази позицију првог елемента који је већи од датог. При том не морамо сваки умањилац обрађивати посебно, већ можемо истовремено обрађивати групе узастопних истих умањилаца (крај групе, тј. број истих елемената можемо одредити поново бинарном претрагом).

Сложеност једне ручно имплементиране бинарне претраге је $O(\log n)$, па пошто се врши $3n$ претрага, сложеност целог решења је $O(n \log n)$.

```
#include <iostream>
#include <algorithm>

using namespace std;

int prviVeciIliJednak(int a[], int l, int r, int x) {
  while (l <= r) {
    int s = l + (r - l) / 2;
    if (a[s] < x)
      l = s + 1;
    else
      r = s - 1;
  }
  return l;
}

int prviVeci(int a[], int l, int r, int x) {
  while (l <= r) {
    int s = l + (r - l) / 2;
    if (a[s] <= x)
      l = s + 1;
    else
      r = s - 1;
  }
  return l;
}

int brojParova(int a[], int n, int d) {
  sort(a, a + n);
  int broj = 0;
  int i = 0;
  while(i < n) {
    int ii = prviVeci(a, i+1, n-1, a[i]);
    int broj_ai = ii - i;
    int j1 = prviVeciIliJednak(a, ii, n-1, a[i] + d);
    int j2 = prviVeci(a, ii, n-1, a[i] + d);
    int broj_aj = j2 - j1;
    broj += broj_ai * broj_aj;
    i = ii;
  }
  return broj;
}


const int N = 50000;
int a[N];
int main() {
  int razlika;
  cin >> razlika;

  int n;
  cin >> n;
```

```
  for (int i = 0; i < n; i++)
     cin >> a[i];

  cout << brojParova(a, n, razlika) << endl;
}
```

Види другачија решења овог задатка.

Задатак: Најближи датом елементу

Сервис за пуштање видео записа корисницима приказује рекламе. За сваку рекламу је познат тренутак (исказан у секундама од почетка видеа) у коме се приказује. Током пуштања корисник може да прескочи на било који тренутак видеа (било унапред било уназад) тј. да пусти видео од било ког места (одређеног секундама од почетка видеа). При том му се приказује реклама и то она која је најближа месту на које је скочио (ако су две рекламе подједнако удаљене од тог места, приказује се она ранија). Напиши програм који одређује која се реклама приказује приликом сваког прескакања видеа.

Улаз: Са стандардног улаза се учитава број реклама n ($1 \leq n \leq 50000$), а затим и n целих бројева (раздвојени размацима) који представљају број секунди од почетка видеа на којем се свака од реклама приказује. Ти бројеви су задати у неопадајућем редоследу (могуће је да се више реклама приказује у истом тренутку). Након тога се уноси број m ($1 \leq m \leq 50000$) који представља број померања током приказивања видео снимка и затим m целих бројева (раздвојени размацима) који представљају број секунди од почетка видеа на који се скаче.

Излаз: На стандардни излаз исписати низ од m бројева који представљају редни број рекламе коју треба приказати (рекламе се броје од 0) за свако померање приказа видеа.

Пример

Улаз	*Излаз*
6	1
1 10 17 17 24 28	0
7	2
9 2 15 26 20 17 35	4
	2
	2
	5

Задатак: Оптимални сервис

Дате су планиране километраже које возач једним камионом треба да пређе у наредних n дана. Елементом a_0 је дата планирана километража за први од n дана, елементом a_1 за други и тако редом. Потребно је извршити сервисирање возила тако да се укупне суме пређених километара пре сервисирања и после сервисирања најмање разликују. Сервисирање се обавља на крају радног дана. Ако постоји два дана са истом најмањом разликом сервис се обавља у ранијем дану. Написати програм којим се одређује на крају ког дана треба извршити сервис.

Улаз: У првој линији стандардног улаза налази се природан број n ($1 < n \leq 50000$). У следећих n линија налазе се по један природан број (између 1 и 1000), бројеви представљају планиране километраже редом за сваки дан.

Излаз: На стандарном излазу приказати у једној линији редни број дана после ког треба вршити сервисирање камиона.

Пример

Улаз	*Излаз*
5	1
100	
120	
50	
150	
70	

Задатак: Први који није дељив

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Линеарна претрага

Наивно решење може бити засновано на примени линеарне претраге (бројању елемената филтриране серије) да би се одредило колико у низу постоји елемената дељивих са учитаним делиоцем.

Ако низ има n елемената, а постоји m делилаца, сложеност овог решења је $O(mn)$. Приметимо да ово решење ни на који начин не користи особину низа да прво иду елементи који су дељиви, а онда елементи који нису дељиви датим делиоцем.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long d;
  while (cin >> d) {
    auto deljiv = [d](long long x) {return x % d == 0;};
    cout << count_if(begin(a), end(a), deljiv) << '\n';
  }
  return 0;
}
```

Бинарна претрага

Захваљујући интересантној особини низа, задатак ефикасно може бити решен применом алгоритма бинарне претраге. У питању је варијанта алгоритма бинарне претраге у ком се уместо позиције конкретне вредности у сортираном низу захтева проналажење прве позиције на којој се налази елемент који задовољава неки услов. Наиме, под претпоставком да се у низу прво налазе елементи који не задовољавају тај услов, а затим елементи који задовољавају тај услов, *преломну тачку* (тренутак када се из једне прелази у другу групу елемената) можемо наћи бинарном претрагом. Дакле, ако је низ облика `-------+++++++`, бинарном претрагом можемо пронаћи позицију последњег минуса, првог плуса, број минуса или број плусева, где смо са `-` означили оне елементе који не задовољавају, а са `+` оне елементе који задовољавају дати услов.

Ручно имплементирана бинарна претрага

Током рада алгоритма, одржавамо две промењиве l и d такве да важи инваријанта да је $0 \leq l \leq d+1 \leq n$ и да су

- лево од l тј. у интервалу позиција $[0, l)$ елементи који не задовољавају услов,
- десно од d тј. у интервалу позиција (d, n) елементи који задовољавају услов.

У интервалу позиција $[l, d]$ налазе се елементи чији статус још није познат. На почетку су сви елементи непознати, па је јасно да интервал $[l, d]$ треба иницијализовати на $[0, n-1]$, тј. промењљиву l треба иницијализовати на нулу, а d на вредност $n-1$. Интервали $[0, l)$ и (d, n) су празни, па је инваријанта очувана (услов $l \leq d+1$ се своди на $0 \leq n$, што је тривијално испуњено).

Ако интервал позиција $[l, d]$ није празан тј. ако је $l \leq d$, проналазимо му средину $s = l + \left\lfloor \frac{d-l}{2} \right\rfloor$.

- Ако елемент на позицији s задовољава услов, тада на основу монотоности услов задовољавају и сви елементи десно од s. Зато померамо d за једно место лево од средине тј. вредност промењиве d постављамо на $s-1$.
- Ако елемент на позицији s не задовољава услов, тада на основу монотоности услов не задовољавају ни сви елементи лево од s. Зато померамо l за једно место десно од средине тј. вредност промењиве l постављамо на $s+1$.

Претрага траје све док се интервал $[l, d]$ не испразни, тј. док је $l \leq d$. Тада је $l = d+1$ и елементи који не задовољавају услов се налазе на позицијама $[0, l) = [0, d]$, док се елементи који не задовољавају услов налазе на позицијама $(d, n) = [d+1, n) = [l, n)$. Дакле, први елемент који задовољава услов је на позицији l, а последњи који не задовољва услов на позицији d.

Прикажимо рад алгоритма на једном примеру.

```
l               d
1 7 3 5 9 11 2 8 6
        s

          l     d
1 7 3 5 9 11 2 8 6
            s

          ld
1 7 3 5 9 11 2 8 6
          s

          d  l
1 7 3 5 9 11 2 8 6
```

Докажимо формално коректност овог алгоритма. Уз поменуте услове инваријанта је и да важи $0 \leq l \leq d+1 \leq n$.

Након иницијализације $l = 0$, $d = n-1$, услови су испуњени (интервали $[0, l)$ и (d, n) су празни, док је услов $0 \leq l \leq d+1 \leq n$ еквивалентан услову $0 \leq 0 \leq n \leq n$ и тривијално је испуњен.

Ако интервал позиција $[l, d]$ није празан тј. ако је $l \leq d$, проналазимо му средину $s = l + \lfloor \frac{d-l}{2} \rfloor$. Пошто је $l \leq d$, важи и да је $l \leq s \leq d$.

- Ако елемент на позицији s задовољава услов, тада на основу монотоности услов задовољавају и сви елементи десно од s. Зато вредност промењиве d постављамо на $s-1$ (нове вредности промењивих су $l' = l$ и $d' = s-1$). Тиме инваријанта остаје на снази (посебно, сви елементи у интервалу позиција $(s-1, n) = [s, n)$ задовољавају услов). Важи и услов $0 \leq l' \leq d'+1 \leq n$, јер је он еквивалентан услову $0 \leq l \leq s \leq n$.
- Ако елемент на позицији s не задовољава услов, тада на основу монотоности услов не задовољавају ни сви елементи лево од s. Зато вредност промењиве l постављамо на $s+1$ (нове вредности промењивих су $l' = s+1$ и $d' = d$). Тиме инваријанта остаје одржана (посебно, ниједан елемент у интервалу позиција $(0, l) = [0, s]$ не задовољава услов). Важи и услов $0 \leq l' \leq d'+1 \leq n$ који је еквивалентан услову $0 \leq s+1 \leq d+1 \leq n$.

Када се интервал испразни, тада је $l > d$, па пошто важи $0 \leq l \leq d+1 \leq n$, важи и $l = d+1$. На основу инваријанте знамо да су елементи који задовољавају услов на позицијама $(d, n) = [l, n]$. Зато је први елемент који задовољава услов је на позиции l (што је уједно и број елемената који не задовољавају услов). Елементи који не задовољавају услов су на позицијама $[0, l) = [0, d+1) = [0, d]$, па је последњи елемент који не задовољава на позицији d.

Заустављање се лако доказује тако што се доказује да се у сваком кораку петље интервал $[l, d]$ тј. његова дужина $d-l+1$ смањује, што је прилично очигледно и када је $l' = l$ и $d' = s-1 < d$ и када је $l < l' = s+1$ и $d' = d$.

Пошто се у сваком кораку претраге ширина интервала $[l, d]$ преполови, пошто се иницијално креће од интервала $[0, n-1]$ који има n елемената и пошто се алгоритам завршава када се интервал испразни, сложеност

алгоритма је $O(\log n)$. Наиме, дужина интервала после k корака је $\left\lfloor \frac{n}{2^k} \right\rfloor$ и важи да је $\left\lfloor \frac{n}{2^k} \right\rfloor < 1$ када је $k > \log_2 n$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long k;
  while (cin >> k) {
    int l = 0, d = n-1;
    while (l <= d) {
      int s = l + (d - l) / 2;
      if (a[s] % k != 0)
        d = s - 1;
      else
        l = s + 1;
    }
    cout << l << '\n';
  }
  return 0;
}
```

Исправљање грешака на основу формалне анализе кода

Када је кôд коректан, доказ је обично неинформативан. Помаже нам да "мирно спавамо", али ништа више од тога. Много интересантнија ситуација се дешава у случају када нам формално резоновање о коду помаже да детектујемо и исправимо грешке у програму (тзв. багове). Погледајмо наредни покушај имплементације алгоритма.

```
  int l = 0, d = n;
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] % k != 0)
      d = s-1;
    else
      l = s+1;
  }
  cout << d+1 << '\n';
```

На основу инцијализације делује да покушавамо да претражимо полузатворени интервал $[l, d)$. Пошто је у питању бинарна претрага, изгледа да се намеће инваријанта да је $0 \leq l \leq d \leq n$ и да су:

- сви елементи из $[0, l)$ дељиви са k,
- ниједан елемент из интервала $[d, n)$ није дељив са k.

На почетку су оба та интервала празна, па инваријанта за сада добро функционише. Ако погледамо услов петље, делује да петља ради док се интервал непознатих елемената $[l, d)$ не испразни (заиста, када је $l \geq d$, тај интервал је празан). За сада све ради како треба. Покушамо сада да проверимо да ли извршавање тела петље одржава инваријанту.

- Ако је a_s није дељив са k, тада се променљива `d` поставља на вредност $d' = s-1$. На основу инваријанте треба да важи да ниједан елемент у интервалу $[d', n)$ није дељив са k. Међутим, ми то не знамо, јер само

знамо да је a_s није дељив са k, али не знамо да a_{s-1} није дељив са k. Дакле, овде се сигурно крије грешка у коду. Ако доделу `d = s-1` заменимо са `d = s`, тада ће инваријанта бити одржана (јер знамо да a_s није дељив са k, па са k неће бити дељив ниједан елемент иза њега).

- Ако a_s јесте дељив са k, тада се променљива `l` поставља на вредност $l' = s + 1$. На основу инваријанте треба да важи да су сви елементи у интервалу $[0, l')$ дељиви са k, међутим, то ће овде бити испуњено, јер је a_s дељив са k, па су са k дељиви и сви елементи испред њега. Дакле, у овом случају је кôд коректан и инваријанта остаје одржана.

На крају, када се петља заврши можемо закључити да важи да је $l = d$ (јер све време важи да је $l \leq d$, а након петље не важи да је $l < d$). У коду се за позицију првог елемента који није дељив са k проглашава позиција $d + 1$. Иако је у оригиналној варијанти кода `l` могло без проблема да се замени са `d+1`, у овој варијанти то није могуће. Наиме, ми на основу инваријанте овог кода знамо да се на позицији $l = d$ налази елемент који није дељив са k, а да се на позицији $l - 1$ налази елемент који јесте дељив са k (осим када је $l = 0$ и тада нема елемената дељивих са k). Зато крајњи резултат није коректан и потребно га је заменити са `d`, јер се први елемент који није дељив са k налази на позицији d (осим када су сви елементи дељиви са k, када је $d = n$, но и тада је d исправна повратна вредност). Дакле, формалном анализом смо открили и исправили две грешке.

Програмери често програм исправљају тако што насумице покушавају да помере индексе за 1 лево или десно, да замене мање са мање или једнако и слично. Већ на овако кратким програмима се види да је простор могућих комбинација велики, а да је могућност за грешку приликом таквог експерименталног приступа веома велика. Стога је увек боље застати, формално анализирати шта је потребно да кôд ради и исправити га на основу резултата формалне анализе.

На крају, скренимо пажњу на још један детаљ исправљеног програма. Парцијална коректност је јасна на основу анализе коју смо спровели, међутим, заустављање може бити доведено у питање, с обзиром на наредбу `d = s`. Заустављање доказујемо тако што показујемо да се у сваком кораку смањује број непознатих елемената, тј. да дужина интервала $[l, d)$ која је једнака $d - l$ у сваком кораку петље опада. Пошто је $l \leq d$ инваријанта, смањивање не може трајати довека, па се у неком тренутку програм зауставља. Поставља се питање да ли се $d - l$ смањује и у измењеном коду у коме се јавља наредба `d=s`. Одговор је потрврдан, а образложење је суптилно. Прво, на основу услова петље важи да је $l < d$. Даље, вредност s се израчунава наредбом `s = l + (d - l) / 2` што нам да је $s = \left\lfloor \frac{l+d}{2} \right\rfloor$. Због заокруживања наниже, важи да је $s < d$ и зато се након одређивања $d' = s$, $l' = l$ вредност $d' - l'$ смањује у односу на $d - l$. Важи и да је $l \leq s$, али пошто је у другој грани $l' = s + 1$ и $d' = d$, вредност $d' - l'$ се опет смањује у односу на $d - l$. Да је заокруживање којим случајем вршено навише (нпр. `s = l + (d - l + 1) / 2`), програм би могао упасти у бесконачну петљу.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long k;
  while (cin >> k) {
    int l = 0, d = n;
    while (l < d) {
      int s = l + (d - l) / 2;
      if (a[s] % k != 0)
        d = s;
      else
        l = s + 1;
    }
    cout << d << '\n';
```

```
  }
  return 0;
}
```

Библиотечке функције

Задатак можемо решити и помоћу библиотечких функција за бинарну претрагу.

Да бисмо нашли први елемент који задовољава неки услов, у језику C++ можемо употребити функцију `upper_bound`, на мало необичан начин. У случају претраге преломне тачке битни су нам само елементи низа, а не елемент који се тражи (јер заправо не тражимо никакав конкретан елемент унутар низа). Зато као елемент који тражимо можемо навести било шта (на пример, нулу). Кључно је дефинисати функцију поређења (која се прослеђује као последњи аргумент функцији `upper_bound`), тако да враћа информацију о томе да су елементи који не задовољавају услов мањи од траженог, док елементи који задовољавају услов нису мањи од траженог. Функција поређења, дакле, треба само да анализира свој други елемент и да врати информацију о томе да ли он задовољава услов (у нашем примеру, тражимо први елемент који није дељив бројем d и то је услов који се проверава у склопу функције поређења).

Рецимо да бисмо могли употребити и функцију `lower_bound`, али би тада у функцији поређења било потребно разменити редослед аргумената (њој је тражена вредност увек други аргумент) и негирати услов.

Пошто је сложеност библиотечке функције бинарне претраге $O(\log n)$, сложеност одговора на свих m упита је $O(m \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long d;
  while (cin >> d) {
    auto it = upper_bound(begin(a), end(a), 0,
                          [d](long long _, long long x) {
                            return x % d != 0;
                          });
    cout << distance(begin(a), it) << '\n';
  }
  return 0;
}
```

Задатак: К-најближих датом

Млади програмер жели да вежба програмирање тако што ће се организовати онлајн такмичење против одређеног броја својих вршњака. Пошто жели да реално процени своје тренутно знање, жели да одабере оне такмичаре који имају рејтинг најсличнији његовом. Ако је приликом одабира противника потребно бирати између такмичара са вишим и нижим рејтингом, потребно је одабрати такмичара са нижим рејтингом. Наш програмер није себичан и жели да његов програм употреби и да помогне свим својим другарима, да одаберу противнике на основу својих рејтинга. Такмичара је пуно и он жели брз одговор.

Улаз: Са стандардног улаза се учитавају рејтинзи потенцијалних противника, сортирани растуће. Сви противници имају различит рејтинг. Прво се учитава број n ($1 \leq n \leq 10000$), а затим и n рејтинга (цели бројеви). Након тога се учитава број m ($1 \leq m \leq 10000$) који представља број такмичења које је потребно

организовати. У наставку се учитава m парова целих бројева, који представљају рејтинг такмичара за кога се организује такмичење и број такмичара у такмичењу (по један пар у сваком реду).

Излаз: На стандардни излаз исписати m бројева (сваки у посебном реду) који представљају индексе првог одабраног противника за свако такмичење (бројање индекса креће од нуле).

Пример

Улаз	*Излаз*
6	0
3	0
5	1
7	3
9	3
11	3
13	
6	
1 3	
6 3	
8 3	
11 3	
13 3	
19 3	

Објашњење

Такмичар са рејтингом 1 треба да се такмичи против такмичара са рејтинзима 3, 5, 7 (индекс првог је 0). Такмичар са рејтингом 6 може да се бори против такмичара са рејтингом 3, 5 и 7 или против такмичара са рејтингом 5, 7 и 9, међутим, приликом избора између такмичара са рејтингом 3 и 9, предност има онај са нижим рејтингом 3, па је индекс првог противника опет 0. Сличан принцип се примењује и у осталим такмичењима.

Решење

Задатак захтева да се у низу сортираних бројева пронађе позиција почетка k-точланог сегмента који садржи елементе најближе датој вредности x.

Проналажење најближег елемента

Један могући приступ је да се прво у низу пронађе елемент најближи вредности x. То је могуће урадити коришћењем бинарне претраге (слично као у задатку Најближи датом елементу), при чему чињеница да су сви елементи низа различити још додатно олакшава задатак. Ако постоје два елемента подједнако удаљена од x, узимамо мањи од њих. Након што знамо позицију најближег елемента вредности x, померамо се од њега на лево и на десно узимајући увек мањи од два потенцијална (ако су леви и десни елемент подједнако удаљенни, узимамо леви), све док тако не прикупимо k тражених елемената.

Најближи елемент вредности x можемо пронаћи у времену $O(\log n)$, али је померање лево и десно од њега сложености $O(k)$, што даје укупну сложеност од $O(m \cdot (\log n + k))$. Ако су m и k приближно једнаки n, ово доводи до укупне квадратне сложености.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <cmath>
#include <iterator>

using namespace std;

int pozicijaNajblizegDatom(const vector<int>& a, int x) {
  auto it = upper_bound(begin(a), end(a), x);
  if (it == end(a) || (it > begin(a) && x - *prev(it) < *it - x))
    it = prev(it);
  return distance(begin(a), it);
}
```

```
int k_najblizih_datom(const vector<int>& a, int x, int k) {
  int p = pozicijaNajblizegDatom(a, x);
  int broj = 1;
  int l = p - 1, d = p + 1;
  while (broj < k) {
    if (l < 0)
      d++;
    else if (d >= a.size())
      l--;
    else if (abs(a[l] - x) <= abs(a[d] - x))
      l--;
    else
      d++;
    broj++;
  }
  return l+1;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    int x;
    cin >> x;
    int k;
    cin >> k;
    cout << k_najblizih_datom(a, x, k) << endl;
  }

  return 0;
}
```

Линеарна претрага

Претпоставимо да је неких k кандидата најбољи избор. Поставља се питање да ли је могуће наћи боље кандидате. Померање за једну позицију удесно елиминише најмањи од тих k елемената и додаје један већи. Тај избор ће бити бољи, само ако је (апсолутна) разлика између x и елемента који се додаје строго мања од (апсолутне) разлике између x и елемента који се избацује. Ако то није случај, померање надесно само може да поквари ситуацију. Дакле, можемо кренути од првих k кандидата и померати се удесно све док то померање побољшава ситуацију. Први пут када наиђемо на кандидате за које важи да је $|x_{i+k} - x| \geq |x_i - x|$, прекидамо поступак и пријављујемо да се решење налази на позицији i. Поступак се прекида и када се дође до краја низа (тј. када је $i + k \geq n$, где је n дужина низа) и тада је решење $n - k$.

Сложеност линеарне претраге је $O(n - k)$, што је у најгорем случају, када је k око $\frac{n}{2}$ једнако $O(n)$, па је укупна сложеност $O(m \cdot n)$, тј. опет квадратна, када је m приближно једнако n.

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;
```

```
int k_najblizih_datom(const vector<int>& a, int x, int k) {
  int i;
  for (i = 0; i < a.size() - k; i++)
    if (abs(a[i] - x) <= abs(a[i+k] - x))
      break;
  return i;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    int x;
    cin >> x;
    int k;
    cin >> k;
    cout << k_najblizih_datom(a, x, k) << endl;
  }

  return 0;
}
```

Бинарна претрага

Ако дефинишемо својство $P(i)$ које важи ако и само ако је $|x_{i+k} - x| \geq |x_i - x|$, важи да на у првом делу низа ово својство није задовољено, све до једног тренутка када постаје и остаје задовољено (све до позиције $n - k - 1$). Тражени индекс је први (најмањи) онај за који је својство P тачно. Ово нам омогућава да применимо бинарну претрагу (на начин описан у задатку Први који није дељив).

Сложеност поступка бинарне претраге је $O(\log(n - k))$, па је укупна сложеност $O(n \log n)$ када је k око $\frac{n}{2}$, а m приближно једнако n.

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

int k_najblizih_datom(const vector<int>& a, int x, int k) {
  int l = 0, d = a.size() - k - 1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (abs(a[s] - x) <= abs(a[s+k] - x))
      d = s - 1;
    else
      l = s + 1;
  }
  return l;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
```

```
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    int x;
    cin >> x;
    int k;
    cin >> k;
    cout << k_najblizih_datom(a, x, k) << endl;
  }

  return 0;
}
```

Библиотечке функције

Имплементацију можемо веома једноставно направити коришћењем библиотечке функције `upper_bound`, како је показано у задатку Први који није дељив. Приступ члановима низа из анонимне функције поређења остварује се на исти начин ка у задатку Разлика висина.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

int k_najblizih_datom(const vector<int>& a, int x, int k) {
  auto it = upper_bound(begin(a), prev(end(a), k), 0,
                        [x, k](int _, const int& y) {
                          return abs(y - x) <= abs(*(next(&y, k)) - x);
                        });
  return distance(begin(a), it);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    int x;
    cin >> x;
    int k;
    cin >> k;
    cout << k_najblizih_datom(a, x, k) << endl;
  }

  return 0;
}
```

Бинарна претрага за интервалом у ком се налази k елемената низа

Задатак можемо решити тако што прво пронађемо најужи интервал вредности облика $(x-r, x+r)$ који садржи бар k елемената низа. То можемо урадити бинарном претрагом по интервалу могућих вредности r. Број r се сигурно налази у интервалу $[0, \max(x-a_0, a_{n-1}-x)]$ (ове границе су коректне и у случају када је x ван граница низа). За фиксирану вредност r број елемената у интервалу $(x-r, x+r)$ можемо пронаћи тако што бинарном претрагом пронађемо најмању позицију i такву да је $a_i > x-r$ и најмању позицију j такву да је $a_j < x+r$.

Када је одређена најмања вредност r, одређујемо интервал позиција $[i, j]$ у низу такав да сви елементи на тим позицијама припадају интервалу $(x-r, x+r)$ (поново i одређујемо као најмању позицију i такву да је $a_i > x-r$, а j као најмању позицију такву да је $a_j < x+r$). Знамо да у интервалу $[i, j]$ има бар k елемената, али можда их има и више од k. Зато интервал сужавамо (елиминишући крајњи леви или крајњи десни елемент који је даљи од x, при чему ако су крајњи елементи на истом растојању од x елиминишемо десни) све док у њему не буде тачно k елемената.

Бинарном претрагом се претражује интервал $[0, \max(x-a_0, a_{n-1}-x)]$ и у сваком кораку се изводи нова бинарна претрага низа a дужине n. Сложеност овог корака је $O(\log(\max(x-a_0, a_{n-1}-x)) \cdot \log n)$. На крају изводе још две бинарне претраге сложености $O(n)$ и затим петља у којој се интервал сужава. Пошто се у низу не понављају елементи, сложеност те петље је $O(1)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

// broj elemenata niza a u intervalu (s, t)
int broj_elemenata_u_intervalu(const vector<int>& a, int s, int t) {
  // prvi strogo veci od s
  auto l = upper_bound(begin(a), end(a), s);
  // poslednji strogo manji od t
  auto d = prev(lower_bound(begin(a), end(a), t));
  return distance(l, d) + 1;
}


int k_najblizih_datom(const vector<int>& a, int x, int k) {
  int n = a.size();
  // binarnom pretragom po vrednosti odredjujemo najmanje r tako da u
  // intervalu (x-r, x+r) ima bar k elemenata niza
  int l = 0, d = max(x - a[0], a[n-1] - x) + 1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (broj_elemenata_u_intervalu(a, x-s, x+s) >= k)
      d = s - 1;
    else
      l = s + 1;
  }
  int r = d + 1;

  // odredjujemo indekse prvog i poslednjeg elementa koji pripadaju
  // intervalu (x-r, x+r)
  auto li = distance(begin(a), upper_bound(begin(a), end(a), x - r));
  auto di = distance(begin(a), prev(lower_bound(begin(a), end(a), x + r)));
  // u intervalu pozicija [li, di] ima bar k elemenata - ako ima vise
  // od k, tada interval treba da se suzi
  while (di - li + 1 > k) {
    if (x - a[li] < a[di] - x)
```

```
      li++;
    else
      di--;
  }

  return li;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    int x;
    cin >> x;
    int k;
    cin >> k;
    cout << k_najblizih_datom(a, x, k) << endl;
  }

  return 0;
}
```

Задатак: Врх планине

Планинари су се пели на планину и сваких 10 минута јављали су надморску висину на којој се налазе. Одредити која је висина планине (ако претпоставимо да су планинари јавили и надморску висину и док су се одмарали на врху).

Улаз: Са стандардног улаза се учитава број n ($3 \leq n \leq 50000$) а затим низ од n различитих бројева таквих да низ прво расте (планинари се пењу), а затим опада (планинари силазе). И лево и десно од висине врха планине постоји бар један број.

Излаз: На стандардни излаз исписати један број који представља висину планине.

Пример

Улаз	*Излаз*
5 3 5 8 6 3	8

Задатак: i-ти на месту i

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Минимум ротираног сортираног низа

Сортирани низ целих бројева у коме су сви елементи различити је ротиран за k места улево и тиме је добијен циклични низ који задовољава услов да је $x_k < x_{k+1} < \ldots < x_{n-1} < x_0 < \ldots < x_{k-1}$. Један такав низ је, на пример, `11 13 15 19 24 1 3 8 9`. Напиши програм који проналази најмањи елемент таквог низа. Потруди се да се након учитавања елемената минимум пронађе у временској сложености $O(\log n)$.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 50000$), а затим n елемената низа (сваки у посебном реду).

Излаз: На стандардни излаз исписати најмањи елемент низа.

Пример

Улаз	*Излаз*
9 11 13 15 19 24 1 3 8 9	1

Задатак: Претрага ротираног сортираног низа

Сортирани низ целих бројева у коме су сви елементи различити је ротиран за k места улево и тиме је добијен циклични низ који задовољава услов да је $x_k < x_{k+1} < \ldots < x_{n-1} < x_0 < \ldots < x_{k-1}$. Један такав низ је, на пример, `11 13 15 19 24 1 3 8 9`. Напиши програм који ефикасно врши претрагу таквог низа.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим n елемената низа (сваки у посебном реду). Након тога се учитава број m ($1 \leq m \leq 50000$). У наредних m редова се учитава m целих бројева који се траже у низу.

Излаз: За сваки од учитаних бројева исписати `0` ако број припада низу тј. `1` ако број не припада низу (сваку цифру исписати у посебном реду).

Пример

Улаз	*Излаз*
5 6 9 1 3 4 3 7 4 2	0 1 0

Решење

Линеарна претрага

Наиван начин да се задатак реши је да се занемари сортираност и да се примени линеарна претрага.

Ово решење је веома неефикасно, јер је сложеност сваке претраге $O(n)$, па је укупна сложеност $O(m \cdot n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  // ucitavamo niz
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  vector<int> a(n);
```

```
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // obradjujemo m upita
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    // linearnom pretragom trazimo ucitani element
    int x;
    cin >> x;
    if (find(begin(a), end(a), x) != end(a))
      cout << 1 << "\n";
    else
      cout << 0 << "\n";
  }

  return 0;
}
```

Две бинарне претраге

Боље решење је да се примени бинарна претрага. На основу задатка Минимум ротираног сортираног низа могуће је бинарном претрагом наћи позицију најмањег елемента у низу. Тиме је низ подељен на два строго растућа сегмента и на сваки од њих је могуће засебно применити класичну бинарну претрагу за траженим елементом.

Претрага сваког елемента захтева две бинарне претраге, па је сложеност претраге сваког елемента $O(\log n)$, а укупна сложеност је $O(m \log n)$ (уз учитавање података у сложености $O(n)$).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  // ucitavamo niz
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // binarnom pretragom odredjujemo poziciju minimuma
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d-l)/2;
    if (a[s] < a[n-1])
      d = s-1;
    else
      l = s+1;
  }
  int min = l;

  // obradjujemo m upita
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    // binarnim pretragama trazimo element u dva sortirana dela niza
```

```
    int x;
    cin >> x;
    if (binary_search(next(begin(a), min), end(a), x) ||
        binary_search(begin(a), next(begin(a), min), x))
      cout << 1 << "\n";
    else
      cout << 0 << "\n";
  }

  return 0;
}
```

Једна бинарна претрага

Могуће је направити и решење које користи само једну бинарну претрагу. Као и увек код бинарне претраге, циљ нам је да једним питањем елиминишемо бар пола елемената низа. Претпоставимо да претражујемо сегмент одређен позицијама $[l, d]$ и нека је s средишња позиција у том сегменту.

Ако се тражени елемент налази на позицији s, претрага је успешна.

У супротном су нам остала два сегмента низа (сегмент одређен позицијама $[l, s-1]$ и сегмент одређен позицијама $[s+1, d]$). Кључни увид је да је бар један од та два дела низа сортиран (а могу бити сортирана и оба). Први сегмент је сортиран ако је $a_l < a_{s-1}$, а други ако је $a_{s+1} < a_d$.

- Ако је сортиран сегмент одређен позицијама $[l, s-1]$, тада проверавамо да ли елемент може бити у њему (то је случај када је $a_l \leq x \leq a_{s-1}$). Ако може, онда смо сигурни да да елемент није у другом делу низа (јер знамо да су сви елементи у сегменту одређеном позицијама $[s+1, d]$ или строго мањи од a_l или строго већи од a_{s-1}) и претрагу сводимо на сегмент $[l, s-1]$. Ако x није у тим границама, онда није у том сегменту, па претрагу сводимо на сегмент одређен позицијама $[s+1, d]$.
- Ситуацију у којој је сортиран сегмент $[s+1, d]$ обрађујемо потпуно аналогно.

Сложеност овог приступа је $O(m \log n)$ (уз учитавање података у сложености $O(n)$).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// proverava da li rotirani sortirani niz a sadrzi element x
bool sadrzi(const vector<int>& a, int x) {
  int n = a.size();
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d-l)/2;
    if (a[s] == x)
      return true;
    if (l < s && a[l] < a[s-1]) {
      // deo [l, s-1] je neprazan i sortiran
      if (a[l] <= x && x <= a[s-1])
        d = s-1;
      else
        l = s+1;
    } else {
      if (s == d)
        // deo [s+1, d] je prazan
        return false;
      // deo [s+1, d] je neprazan i sortiran
      if (a[s+1] <= x && x <= a[d])
        l = s+1;
      else
```

```
        d = s-1;
    }
  }
  return false;
}

int main() {
  // ucitavamo ulazni niz
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // obradjujemo m upita
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    // prilagodjenom binarnom pretragom trazimo ucitani element
    int x;
    cin >> x;
    if (sadrzi(a, x))
      cout << 1 << "\n";
    else
      cout << 0 << "\n";
  }

  return 0;
}
```

Сортирање и класична бинарна претрага

Још једно могуће решење је да се низ након учитавања сортира и да се примењује класична бинарна претрага. Низ се може сортирати било класичним сортирањем, најбоље библиотечком функцијом, било проналажењем позиције минимума (слично као у задатку Минимум ротираног сортираног низа) и ротирањем.

Класично сортирање низа је сложености $O(n \log n)$, док се минимум може пронаћи у времену $O(\log n)$, а затим се низ може ротирати у времену $O(n)$. Након тога се m пута врши класична бинарна претрага, сложености $O(\log n)$. Укупна сложеност ако се користи класично сортирање је $O((n+m)\log n)$, а ако се користи ротирање $O(n + m \log n)$.

Постоји могућност и да се ангажује двоструко више меморије, да се низ два пута учита и затим да се проналажењем минимума одреди сортирани распон који садржи елементе оригиналног низа и који се може претраживати сасвим класичном бинарном претрагом.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  // ucitavamo niz
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
```

```
  // sortiramo niz
  sort(begin(a), end(a));

  // obradjujemo m upita
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    // binarnim pretragama trazimo element u dva sortirana dela niza
    int x;
    cin >> x;
    if (binary_search(begin(a), end(a), x))
      cout << 1 << "\n";
    else
      cout << 0 << "\n";
  }

  return 0;
}
```

2.9.3 Проналажење оптималне вредности решења бинарном претрагом

Бинарна претрага се може употребити и у процесу оптимизације, ако се проблем може формулисати као проблем проналажења преломне тачке. Овај облик претраге се понекад назива *Бинарна претрага по решењу*. Идеја је да се проблем оптимизације "наћи најмању вредност која задовољава одређени услов", сведе на проблем одлучивања "да ли дата вредност задовољава одређени услов". Бинарну претрагу је могуће применити ако проблем задовољава својство монотоности, које захтева да ако нека вредност задовољава услов, онда услов задовољавају и све вредности веће од ње, а ако не задовољава, онда услов не задовољавају ни вредности мање од ње. Наравно, сасвим сличан је задатак проналажења највеће вредности која не задовољава услов. Карактеристично за ову употребу бинарне претраге је то што вредности о којима је реч обично нису индекси елемената низа, а често се врши оптимизација и над непрекидним скупом вредности (до на одређену тачност). Такође, провера испуњења услова за сваку конкретну вредност је обично спора и желимо да смањимо број провера испуњења услова колико је могуће. Стога се уместо коришћења библиотечких функција, бинарна претрага ручно имплементира.

Задатак: Дрва

Дрвосеча треба да насече одређену количину дрвета и има тестеру коју може да подешава да сече на било којој целобројној висини (у метрима). Пошто тестера сече само дрво изнад висине на коју је постављена, што је тестера више, насећи ће се мање дрвета. Пошто дрвосеча брине о околини, он не жели да насече више дрвета него што му је потребно. Напиши програм који одређује највишу могућу целобројну висину тестере, тако да дрвосеча добије довољно дрвета (претпостави да увек постоји довољно дрвета).

Улаз: Са стандардног улаза се учитава број дрвећа у шуми n ($1 \leq n \leq 10^5$), а затим низ висина сваког дрвета (низ природних бројева између 1 и 10000, сваки у посебном реду). Након тога учитава се и количина насеченог дрвета (пошто су сва дебла исте дебљине, количина се мери у метрима висине исечених стабала).

Излаз: На стандардни излаз исписати тражену максималну целобројну висину тестере.

Пример

Улаз	*Излаз*
5 24 21 19 14 22 14	18

Решење

Оптимизација бинарном претрагом

Једно решење проблема се може засновати на бинарној претрази по решењу тј. по тражењу оптималне вредности коришћењем бинарне претраге. Постављањем тестере на висину h, код свих дрва која су виша од h биће одсечено $h_i - h$ метара, док од осталих дрва неће бити исечено ништа. На основу тога, за фиксирану висину тестере грубом силом (испитивањем сваког дрвета засебно) у времену $O(n)$ можемо израчунати укупну количину насеченог дрвета. Бинарна претрага је применљива јер знамо да је до одређених висина тестере дрвета довољно, а да је од одређене висине тестере дрвета премало, тако да заправо тражимо преломну тачку, тј. највећу висину тестере за коју је дрвета довољно тј. последњи елемент низа који задовољава услов. Техника како се то може урадити описана је у задатку Први који није дељив. Приметимо да у овом случају немамо вредности смештене у низ, већ их рачунамо по потреби. Зато бинарну претрагу имплементирамо ручно.

Ако је максимална висина дрвета M, тада је сложеност овог приступа $O(n \log M)$. Наиме, бинарном претрагом се претражује интервал $[0, M]$, па се провера да ли је насечено довољно дрвета позива $\log M$ пута. Израчунавање количине насеченог дрвета и провера да ли је она довољна врши се једним пролазак кроз низ дрвета и сложености је $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int testera(const vector<int>& visine, int potrebno) {
  int od_visina = 0;
  int do_visina = *max_element(begin(visine), end(visine));
  while (od_visina <= do_visina) {
    int visina = od_visina + (do_visina - od_visina) / 2;
    long long naseceno = 0;
    for (int v : visine)
      if (v >= visina)
        naseceno += v - visina;

    if (naseceno >= potrebno)
      od_visina = visina + 1;
    else
      do_visina = visina - 1;
  }
  return do_visina;
}

int main() {
  int n;
  cin >> n;
  vector<int> visina(n);
  for (int i = 0; i < n; i++)
    cin >> visina[i];
  long long potrebno;
  cin >> potrebno;

  cout << testera(visina, potrebno) << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Конференција

Током једног дана конференције одржава се n предавања. Представља се m рачунарских компанија које су означене бројевима од 1 до m, при чему неке компаније држе и више предавања. Један од учесника конференције жели да присуствује предавањима свих компанија без напуштања сале за предавања. Написати

програм који одређује који је најмањи број предавања које мора да одслуша да би чуо све компаније.

Улаз: На стандардном улазу у првом реду је број компанија m ($1 \leq m \leq 50$), у другом реду је укупан број предавања n ($1 \leq n \leq 50000$), а у трећем је наведен редослед предавања компанија тако што су наведени њихови редни бројеви (од 1 до m) раздвојени размацима.

Излаз: Најмањи број предавања које учесник треба да одслуша у континуитету да би чуо бар по једно предавање сваке компаније.

Ако није могуће да учесник одслуша предавање сваке компаније, исписати текст “ne moze” (без наводника).

Пример 1

Улаз

```
4
20
4 2 4 3 3 2 2 4 2 2 3 3 1 3 3 1 4 4 1 4
```

Излаз

```
6
```

Објашњење

Могуће је одслушати предавања компанија `4 2 2 3 3 1`

Пример 2

Улаз

```
3
10
1 2 1 2 1 2 1 2 1 2
```

Излаз

```
ne moze
```

Решење

Груба сила

Решење грубом силом подразумева да се испитају сви сегменти. За сваки сегмент испитујемо да ли су у њему одслушана предавања свих компанија (за сваку компанију линеарном претрагом испитујемо да ли се у тренутном сегменту налази ознака те компаније).

Пошто сегмената има $O(n^2)$, компанија је m, а претрага сегмента да би се проверила да ли се у њему налази компанија је линеарна у односу на дужину сегмента, овим добијамо алгоритам сложености $O(m \cdot n^3)$, што је страшно неефикасно.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int broj_kompanija;
  cin >> broj_kompanija;

  int broj_predavanja;
  cin >> broj_predavanja;
  vector<int> predavanja(broj_predavanja);
  for (int i = 0; i < broj_predavanja; i++)
    cin >> predavanja[i];

  // proveravamo sve segmente oblika [i, j)
  int min = broj_predavanja + 1;
  for (int i = 0; i + broj_kompanija <= broj_predavanja; i++)
    for (int j = i + broj_kompanija; j <= broj_predavanja; j++) {
      bool slusao_sve = true;
```

```
      for (int k = 1; k <= broj_kompanija; k++) {
        auto pocetak = next(begin(predavanja), i);
        auto kraj = next(begin(predavanja), j);
        if (find(pocetak, kraj, k)  == kraj) {
          slusao_sve = false;
          break;
        }
      }
      if (slusao_sve && j - i < min)
        min = j - i;
    }

  if (min == broj_predavanja + 1)
    cout << "ne moze" << endl;
  else
    cout << min << endl;

  return 0;
}
```

Оптимизације решења грубом силом

Основно решење се може унапредити разним техникама. На пример, не морамо испитивати све сегменте који почињу на текућој левој позицији, већ само оне који су краћи од досадашњег минимума, проверу да ли текући сегмент садржи неку компанију можемо вршити инкрементално, тако што одржавамо скуп компанија које држе предавање у текућем сегменту (слично као у задатку Панграми) и слично.

Сложеност најгорег случаја свих тих решења биће бар $O(n^2)$ што је и даље недопустиво неефикасно.

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <algorithm>

using namespace std;

int main() {
  size_t broj_kompanija;
  cin >> broj_kompanija;

  size_t broj_predavanja;
  cin >> broj_predavanja;
  vector<int> predavanja(broj_predavanja);
  for (size_t i = 0; i < broj_predavanja; i++)
    cin >> predavanja[i];

  // proveravamo sve segmente oblika [i, j)
  size_t min = broj_predavanja + 1;
  for (size_t i = 0; i + broj_kompanija <= broj_predavanja; i++) {
    // skup kompanija koje je posetilac slusao unutar tekuceg segmenta
    unordered_set<int> slusao;
    for (size_t j = i + 1; j <= broj_predavanja && j < i + min; j++) {
      slusao.insert(predavanja[j-1]);
      // odslusane su sve kompanije
      if (slusao.size() == broj_kompanija) {
        // nasli smo kraci segment, pa azuriramo minimum
        min = j - i;
        break;
      }
    }
```

```
  }

  if (min == broj_predavanja + 1)
    cout << "ne moze" << endl;
  else
    cout << min << endl;

  return 0;
}
```

Оптимизација бинарном претрагом

За дати број узастопних предавања d, једним проласком кроз низ можемо проверити да ли је могуће да се током неких d узастопних предавања чују све компаније. Можемо одржавати мултискуп компанија чија су предавања одслушана током текућег низа узастопних предавања и проверити да ли је број различитих компанија у том мултискупу једнак укупном броју компанија. Скуп можемо представити било библиотечким колекцијама (мапом тј. речником), било низом бројача. Приликом преласка са једног, на наредни низ узастопних предавања, скуп ажурирамо инкрементално, избацујући прво предавање старог низа и убацујући последње предавње новог низа.

Када на располагању имамо проверу да ли је могуће за неки низ узастопних предавања дужине d одслушати сва предавања, тада бинарном претрагом лако можемо наћи најмању вредност d за коју је тај услов испуњен. Пошто је низ потенцијалних вредности d само имплицитан, претрагу реализујемо ручно (по принципу тражења преломне тачке, како је описано у задатку Први који није дељив).

Бинарном претрагом се претражује интервал $[1, n-1]$. Провера да ли су све компаније заступљене врши се инкрементално, једним проласком кроз низ предавања дужине n. У сваком кораку се врши и приступ мапи тј. речнику чији су кључеви компаније. Ако претпоставимо да су операције над мапом логаритамске сложености, једна провера се врши у времену $O(n \log m)$, па пошто тих провера има највише $\log n$, укупна сложеност је $O(n \log n \log m)$. Пошто је број компанија јако мали број, фактор $\log m$ можемо сматрати и малом константном, па се сложеност може грубо проценити са $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

// provera da li postoji neki niz uzastopnih predavanja date duzine u
// kom se mogu cuti sve kompanije
bool sveKompanije(const vector<int>& predavanja, int broj_kompanija, int duzina) {
  // broj predavanja svake kompanije u tekucem nizu
  map<int, int> kompanije;
  // obradjujemo sva predavanja
  for (int i = 0; i < predavanja.size(); i++) {
    // pri prelasku na sledeci niz, uklanjamo prvo predavanje
    // prethodnog niza
    if (i >= duzina)
      if (--kompanije[predavanja[i-duzina]] == 0)
        kompanije.erase(predavanja[i-duzina]);
    // dodajemo novo predavanje i proveravamo da li smo culi sve
    // kompanije
    kompanije[predavanja[i]]++;
    if (kompanije.size() == broj_kompanija)
      return true;
  }
  return false;
}

int main() {
  // ucitavamo podatke o kompanijama i predavanjima
```

```
  int broj_kompanija;
  cin >> broj_kompanija;
  int broj_predavanja;
  cin >> broj_predavanja;
  vector<int> predavanja(broj_predavanja);
  for (int i = 0; i < broj_predavanja; i++)
    cin >> predavanja[i];

  // binarnom pretragom nalazimo minimalni niz uzastopnih predavanja u
  // kom se mogu cuti sve kompanije
  int l = 1, d = broj_predavanja - 1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (sveKompanije(predavanja, broj_kompanija, s))
      d = s-1;
    else
      l = s+1;
  }

  if (l == broj_predavanja)
    cout << "ne moze" << endl;
  else
    cout << l << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Најкраћа подниска која садржи све дате карактере

Графички дизајнер је преуредио неколико слова у једном фонту и жели да своје промене прикаже клијенту. У дугачком тексту је потребно да одабере најкраћи део (сегмент узастопних слова) који садржи сва слова која је променио.

Улаз: У првој линији стандардног улаза налази се текст (једноставности ради претпоставимо да је састављен само од малих слова енглеског алфабета) чија је дужина највише 50000 карактера. У другој линији се налази скуп слова (опет, претпоставимо малих слова енглеског алфабета) које је дизајнер променио (слова су написана једно до другог, без размака и без понављања).

Излаз: На стандардни излаз исписати један цео број који представља дужину најкраћег дела текста који садржи све карактере датог скупа. Ако такав део текста не постоји, исписати `nema`.

Пример 1

Улаз	*Излаз*
`dobardansvimakakoste` `arnk`	`10`

Пример 2

Улаз	*Излаз*
`abababababab` `abc`	`nema`

Решење

Овај задатак представља једноставно уопштење задатка Конференција.

Груба сила - провера свих подниски

Најдиректнији алгоритам био би да се разматрају сви подтекстови (они су одређени индексима $0 \leq i \leq j < n$), да се за сваки подтекст провери да ли је исправан тј. да ли садржи све карактере из скупа S и да се међу свим исправним пронађе најкраћи. Овакву претрагу грубом силом је веома једноставно имплементирати.

Сложеност је $O(n^3 \cdot m)$ где је n дужина текста, а m број карактера у скупу S. Параметар m има малу вредност, али димензија n може значајно да расте, па је пожељно пронаћи бољи алгоритам.

```
#include <iostream>
#include <string>
#include <limits>
```

```cpp
using namespace std;

// proverava da li niska T[i, j] sadrzi karakter c
bool podniskaSadrziKarakter(const string& T, int i, int j, char c) {
  for (int k = i; k <= j; k++)
    if (T[k] == c)
      return true;
  return false;
}

int main() {
  string T, S;
  getline(cin, T);
  getline(cin, S);

  // duzina najkrace do sada pronadjene niske koja sadrzi sve
  // karaktere iz S
  int min_duzina = numeric_limits<int>::max();

  // obradjujemo sve pozicije pocetka podniske
  for (int i = 0; i < T.size(); i++) {
    // preskacemo karaktere koji nisu u skupu S
    // jer najkraca podniska mora da pocne sa karakterom iz S
    if (S.find(T[i]) == string::npos) continue;

    // obradjujemo sve pozicije kraja podniske koja pocinje na poziciji i
    for (int j = i; j < T.size(); j++) {
      // proveravamo da li podniska T[i, j] sadrzi sve karaktere iz S
      bool nedostaje = false;
      for (char c : S)
        if (!podniskaSadrziKarakter(T, i, j, c)) {
          nedostaje = true;
          break;
        }
      if (!nedostaje) {
        int duzina = j - i + 1;
        if (duzina < min_duzina)
          min_duzina = duzina;
        break;
      }
    }
  }

  // prijavljujemo rezultat
  if (min_duzina != numeric_limits<int>::max())
    cout << min_duzina << endl;
  else
    cout << "nema" << endl;
}
```

Груба сила - оптимизације

Могући корак оптимизације може бити да се примети да када сегмент одређен позицијама i и j садржи све карактере из скупа S, онда и сви сегменти одређени већим вредностима j такође садрже све карактере из скупа, а за њих знамо да су дужи и нема потребе разматрати их.

На пример, ако се у речи `xCxxBxxBxxAxxBxxCxxxxBxAxAxCxxxBx` тражи подтекст у коме се јављају слова из скупа `ABC`. Након проналаска подтекста `CxxBxxBxxA`, који садржи све потребне карактере, нема потребе разматрати његове суфиксе (на пример, подтекст `CxxBxxBxxAxxB`).

Дакле први пронађен исправан подтекст који почиње на позицији i уједно је и најкраћи исправан подтекст пронађен на тој позицији. Зато је одмах након проналаска првог исправног подтекста и ажурирања вредности најмање дужине могуће прекинути унутрашњу петљу (наредбом `break`).

Ова оптимизација не поправља асимптотску сложеност најгорег случаја.

Позиције карактера из скупа

Прво приметимо да најкраћи подтекст мора да почиње и да се завршава карактером из скупа S. У супротном би карактери на почетку и на крају подтекста који нису у скупу S могли да буду уклоњени чиме би се добио краћи подтекст који би и даље садржао све карактере из скупа S. Дакле, приликом разматрања подтекстова потребно је разматрати само карактере из скупа S (рећи ћемо да су само ти карактери релевантни), па ћемо у првој фази само изградити низ (вектор, листу) њихових позиција у тексту. Тада је потребно разматрати само сегменте који почињу и завршавају се на позицијама унутар тог вектора, што у случају да постоји значајан број карактера у тексту који нису у скупу S може убрзати претрагу.

Време потребно за изградњу низа релевантних позиција је $O(n \cdot m)$ (пошто је m мали број, ово је практично линеарно, а може се снизити и на $O(m+n)$ ако би се скуп S представио својом карактеристичном функцијом - асоцијативним низом 26 логичких вредности).

Скуп карактера текућег сегмента

Проверу да ли се сви карактери из скупа S јављају у сегменту између две релевантне позиције i и j можемо извршити тако што одредимо скуп свих карактера на свим релевантним позицијама између i и j. Сви карактери из тако направљеног скупа ће бити у скупу S, јер разматрамо само релевантне позиције тј. само позиције на којима смо претходно установили да се налазе карактери из S, па је уместо испитивања једнакости два скупа, довољно установити само да ли имају исти број елемената (а пошто ниска S по условима задатка нема понављених карактера, број елемената скупа S једнак је дужини ниске). Изградњу скупа карактера који су се појавили можемо вршити инкрементално (слично као у задатку Панграми), тако што приликом сваког повећања j, тј. приликом преласка на нову релевантну позицију j додамо карактер на тој позицији у тај скуп, што захтева само константно време (ако скуп карактера представимо низом или библиотечким скупом, јер је укупан број могућих карактера само 26).

Зато, ако и даље вршимо испитивање свих почетних позиција сложеност постаје $O(n^2)$.

```
#include <iostream>
#include <string>
#include <vector>
#include <limits>
#include <set>

using namespace std;

int main() {
  string T, S;
  getline(cin, T);
  getline(cin, S);

  // duzina najkrace do sada pronadjene niske koja sadrzi sve
  // karaktere iz S
  int min_duzina = numeric_limits<int>::max();

  vector<int> poz_karaktera_iz_S;
  for (int i = 0; i < T.size(); i++)
    // ako se T[i] nalazi u skupu S
    if (S.find(T[i]) != string::npos)
      // zapamti njegovu poziciju i
      poz_karaktera_iz_S.push_back(i);

  for (auto i = poz_karaktera_iz_S.begin(); i != poz_karaktera_iz_S.end(); i++) {
    set<char> karakteri_u_podtekstu;
    for (auto j = i; j != poz_karaktera_iz_S.end(); j++) {
```

```
      karakteri_u_podtekstu.insert(T[*j]);
      if (karakteri_u_podtekstu.size() == S.size()) {
        int duzina = *j - *i + 1;
        if (duzina < min_duzina)
          min_duzina = duzina;
        break;
      }
    }
  }

  // prijavljujemo rezultat
  if (min_duzina != numeric_limits<int>::max())
    cout << min_duzina << endl;
  else
    cout << "nema" << endl;
}
```

Бинарна претрага

Захваљујући инкременталности, проверу да ли за дату дужину подниске k постоји нека подниска која садржи све дате карактере можемо урадити у линеарном времену $O(n)$, у једном проласку кроз низ. Ово додатно можемо убрзати (додуше не асимптотски) ако је скуп карактера мали тако што ћемо пролазити само кроз позиције оригиналне ниске на којима знамо да се јављају карактери из скупа.

Након тога, најмању дужину k можемо наћи техником бинарне претраге тако што нађемо најмању вредност k за коју важи неки услов (слично како је описано у задатку Први који није дељив). Битно је нагласити да проблем задовољава својство монотоности тј. да ако за неко k не постоји подниска дужине k која садржи све карактере скупа, онда таква подниска не постоји ни за једно мање k тј. да ако за неко k постоји таква подниска, онда она постоји и за свако веће k.

Бинарном претрагом се претражује интервал $[1, n]$, где је n дужина ниске T, па се провера да ли постоји подниска неке фиксне која садржи сва слова врши највише $\log n$ пута. Та провера се врши једним проласком кроз ниску T. У сваком кораку врши се линеарна претрага ниске S и врше се операције над мапом тј. речником у ком су кључеви карактери из S. Ако претпоставимо да се операције над мапом извршавају у логаритамској сложености, сложеност једне претраге је зато $n \cdot m \cdot \log m$, где је m број карактера у S. Пошто је m веома мали број, фактор $m \log m$ можемо сматрати и константом и укупну сложеност грубо оценити са $O(n \log n)$. Да је m веће, сложеност би се могла поправљати тиме што би се провера припадности карактера скупу S вршила на ефикаснији начин (на пример, представљањем S помоћу бибилотечких колекција за представљање скупа или помоћу низа логичких вредности). Такође је могуће унапред одредити позиције карактера из S унутар (што не утиче на асимптотску сложеност најгорег случаја, јер сви карактери у T могу припадати S, али може доста смањити фактор n у свакој појединачној провери).

```
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <limits>

using namespace std;

// provera da li postoji podniska niske T duzine k koja sadrzi sve karaktere iz S
bool postojiPodniska(const string& T, const string& S, int k) {
  map<char, int> karakteri;
  for (int i = 0; i < T.length(); i++) {
    if (i >= k && S.find(T[i-k]) != string::npos)
      if (--karakteri[T[i-k]] == 0)
        karakteri.erase(T[i-k]);
    if (S.find(T[i]) != string::npos) {
      karakteri[T[i]]++;
      if (karakteri.size() == S.size())
        return true;
```

```
    }
  }
  return false;
}

int main() {
  string T, S;
  getline(cin, T);
  getline(cin, S);

  // binarnom pretragom odredjujemo najmanju duzinu k takvu da u T
  // postoji podniska duzine k koja sadrzi sve karaktere skupa S
  int l = 1, d = T.length();
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (postojiPodniska(T, S, s))
      d = s - 1;
    else
      l = s + 1;
  }

  // prijavljujemo rezultat
  if (l <= T.length())
    cout << l << endl;
  else
    cout << "nema" << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Хиршов h-индекс

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Види другачија решења овог задатка.

Задатак: Највећи квадрат у хистограму

Напиши програм који одређује површину највећег квадрата који се може уписати у задати хистограм (хистограм се састоји од стубаца ширине 1).

Улаз: Са стандардног улаза се уноси број стубаца n ($1 \leq n \leq 50000$), а затим и висине стубаца (позитивни цели бројеви мањи од 10^5, раздвојени размацима).

Излаз: На стандардни излаз исписати тражену површину.

Пример

Улаз	*Излаз*
5 1 5 4 4 2	9

Задатак: Муцајући подниз

Ако је s ниска, онда нека s^n означава ниску која се добија ако се свако слово понови n пута (нпр. $(xyz)^3$ је $xxxyyyzzz$). Напиши програм који одређује највећи број n такав да је s^n подниз дате ниске t (то значи да се сва слова ниске s^n јављају у ниски t, у истом редоследу као у s^n, али не обавезно узастопно).

Улаз: У првом реду стандардног улаза налази се ниска s, а у другом ниска t.

Излаз: На стандардни излаз напиши тражени број n.

Пример

Улаз	*Излаз*
`xyz` `xaxxybyxyxzyzzb`	`3`

Задатак: Пуно фигурица

Друштвену игру игра k играча и сваки играч игру почиње са по k фигура, при чему све фигуре једног играча морају бити исте јачине, док су јачине фигура различитих играча различите. Фигуре су доступне у неограниченим количинама, при чему је познат низ јачина доступних фигура. Напиши програм који одређује највећи број играча k који могу играти игру тако да разлика између укупне јачине свих фигура било која два играча не пређе задату границу.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим у наредном реду n различитих расположивих јачина фигура (природни бројеви између 1 и 10^5). Наредни ред садржи границу (природни број између 1 и 10^{12}).

Излаз: На стандардни излаз исписати два броја раздвојена размаком – број k и најмању разлику укупних јачина фигура када игра k играча.

Пример

Улаз	*Излаз*

`5`
`5 4 2 7 3`
`15`
	`4 12`

Објашњење

Ако играчи узму по четири фигуре јачине 5, 4, 2 и 3 највећа разлика јачина фигура играча биће $4 \cdot 5 - 4 \cdot 2 = 12$. Ако би играло 5 играча, морали би да узму и фигуре јачине 7 и највећа разлика би била $5 \cdot 7 - 5 \cdot 2 = 25$, што је веће од дозвољене границе.

Види другачија решења овог задатка.

Задатак: Кувар

Кувар жели да гостима на прослави рођендана спреми свој омиљени специјалитет, који се састоји од n састојака. За припрему једне порције потребна је одређена количина сваког састојка (изражена као маса, у грамима). Одређене количине неких састојака већ постоје у кухињи, остатак треба купити. У продавници се продају сви састојци, а сваки састојак је доступан у мањем или већем паковању. Кувар има d динара и жели да их потроши тако да од укупне количине састојака може да се направи што више порција његовог омиљеног јела.

Улаз: У првом реду стандардног улаза се налази број динара d ($1 \leq d \leq 10^7$). У другом реду стандардног улаза налази се број састојака n ($1 \leq n \leq 100$). У наредних n линија се налази по шест природних бројева:

- M ($10 \leq M \leq 100$) - маса састојка потребна за једну порцију (у грамима);
- M_0 ($1 \leq M_0 \leq 100$) - маса састојка која већ постоји у кухињи (у грамима);
- m_m ($1 \leq m_m < 100$) - маса мањег паковања састојка (у грамима);
- c_m ($10 \leq c_m < 100$) - цена мањег паковања састојка (у динарима);
- m_v ($m_m < m_v \leq 100$) - маса већег паковања састојка (у грамима);
- c_v ($c_m < c_v \leq 100$) - цена већег паковања састојка (у динарима).

Излаз: На стандардни излаз исписати највећи број порција које кувар може да припреми са новцем који располаже.

Пример

Улаз	*Излаз*

`100`
`2`
`10 8 10 10 13 11`
`12 20 6 10 17 24`
	`5`

Објашњење

За 99 динара кувар ће купити 3 мања и једно веће паковање првог састојка и једно мање и два већа паковања другог састојка ($3 \cdot 10 + 1 \cdot 11 + 1 \cdot 10 + 2 \cdot 24 = 99$).

Кувар ће тада имати на располагању 51 грам ($8 + 3 \cdot 10 + 1 \cdot 13$) првог састојка и 60 грама ($20 + 1 \cdot 6 + 2 \cdot 17$) другог састојка, што је довољно за 5 порција.

Задатак: Градња

За једну дугачку улицу познате су дозвољене локације за градњу. Свака локација је задата по једним целим бројем, растојањем локације од почетка улице. Потребно је изградити N зграда, али тако да суседне зграде буду што даље једна од друге. Инвеститори оцењују квалитет плана градње G (то јест избора N локација) помоћу растојања $D(G)$. Растојање $D(G)$ представља најмање растојање између узастопних зграда при плану градње G. Инвеститоре посебно интересује која је највећа могућа вредност $D(G)$. Напиши програм који за дате дозвољене локације и потребан број кућа исписује највеће растојање које се може постићи.

Улаз: У првом реду дат је број планираних зграда N ($2 \leq N \leq L$), а у другом број расположивих локација L ($2 \leq L \leq 10^5$). У наредних L редова дати су положаји потенцијалних локација за градњу X_i, ($1 \leq i \leq L$, $0 \leq X_i \leq 10^9$).

Излаз: На стандардни излаз исписати највеће могуће минимално растојање између суседних кућа.

Пример

Улаз	*Излаз*
3 7 9 4 6 2 10 14 12	5

Објашњење

Дато је 7 локација на којима се граде 3 куће. Ако се оне изграде на локацијама 2, 9 и 14, најмање растојање између кућа је 5 (боље решење од овог није могуће).

Задатак: Гласници

Дуж једног пута налазе се гласници. Поруку први сазнаје гласник на почетку пута и циљ је да сви гласници што пре сазнају поруку. Сваки гласник може викањем пренети поруку свима који се од њега налазе на растојању мањем од задатог домета гласа (истог за све гласнике). При том се сви гласници могу кретати брзином од највише једног метра у секунди у било ком смеру (током времена могу мењати своју брзину и смер кретања, а могу и стајати у месту).

Улаз: Са стандардног улаза се учитава домет гласа d (реалан број, $1 \leq d \leq 10^6$), затим број гласника n (цео број, $1 \leq n \leq 10^5$), а након тога положај сваког гласника g_i (реалан број, $0 \leq g_i \leq 10^9$, који представља растојање од почетка пута).

Излаз: На стандардни излаз исписати најмање време протекло од тренутка када први гласник сазна поруку до тренутка када поруку сазнају сви гласници, као реалан број заокружен на три децимале (одступање од тачне вредности сме да буде највише 10^{-3}).

Пример 1

Улаз	*Излаз*
3.000 2 0.000 6.000	1.500

Пример 2

Улаз	*Излаз*
2.000 4 0.000 4.000 4.000 8.000	1.000

Решење

Свођење на проблем одлучивања

Ако је познато време дозвољено за пренос свих порука, можемо формулисати грамзиви алгоритам којим проверавамо да ли је пренос порука могућ у том времену.

Позицију g сваког гласника одређујемо тако да се након времена t налази што даље од почетка пута, али тако да током времена t успешно прима поруку од претходног гласника. Наиме, што је даље од почетка пута након што прими поруку, то ће раније моћи да пренесе поруку онима који се налазе десно од њега. Заиста, ако поруку може да пренесе следећем из неке тачке која је лево од те најдаље, то ће моћи да уради и из те најдаље тачке.

Употребимо индуктивну конструкцију. Разматрамо једног по једног гласника (сваки од њих ће пренети поруку оном ко се налази непосредно иза њега). Претпоставимо да се након времена t гласник i који је током тог времена успешно примио поруку налази на некој позицији G_i и да је то најдаља позиција од почетка пута на којој он може бити тако да током времена t успешно прими поруку.

- За првог гласника то ће бити позиција $G_0 = g_0 + t$.
- За остале гласнике позиције G_{i+1} одређујемо на основу позиција G_i њима претходних гласника.

  Са позиције G_i гласник i ће покушати да пренесе поруку наредном гласнику $i + 1$. Једини начин да то не може да се деси је да му је наредни гласник предалеко тј. да се налази толико десно од њега да се, крећући се све време улево, након времена t налази у тачки $g_{i+1} - t$, а која је десно од граничне тачке домета $G_i + d$. Дакле, ако је $G_i + d < g_{i+1} - t$, порука не може бити пренета.

  Уколико се било када током времена t гласник $i + 1$ нађе унутар домета гласника i, он може све време да остане у домету (крећући се на исти начин као претходи гласник). Дакле, ако је $G_i + d \leq g_{i+1} - t$, порука бива пренета и покушавамо да одредимо положај G_{i+1}. Та тачка не може бити десно од $G_i + d$, јер тада гласник $i + 1$ не би могао да прими поруку. Поставља се питање да ли је могуће да буде баш на позицији $G_i + d$. Једини случај у коме то није могуће је када се гласник i+1 налази превише лево од те тачке тј. када је $g_{i+1} + t < G_i + d$ и тада је $G_{i+1} = g_{i+1} + t$. Дакле, $G_{i+1} = \min(g_{i+1} + t, G_i + d)$.

Приметимо да смо у сваком кораку гласнике терали што више десно, да бисмо им повећали шансу да пренесу поруку. Стога је овај алгоритам грамзив.

Линеарна претрага

Када на располагању имамо функцију за проверу да ли је поруку могуће пренети у датом времену, наиван начин је да време мало по мало повећавамо (узевши у обзир тражену прецизност решења), све док се први пут не догоди да је поруку могуће пренети.

Време потребно да се провери да ли сви гласници могу да приме поруку током времена t захтева један пролаз кроз низ гласника и износи $O(n)$. Број позива функције зависи од броја децимала k (и то фактором 10^k) и од оптималне вредности времена, а оно је ограничено растојањем између првог по последњег гласника. Ако је то растојање X, тада се сложеност најгорег случаја може проценити на $O(10^k \cdot X \cdot n)$.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>

using namespace std;

bool moguceJeRasiritiVest(const vector<double>& pozicije,
                          double domet, double vreme) {
  double p = pozicije[0] + vreme;
  for (int i = 1; i < pozicije.size(); i++) {
    if (p + domet < pozicije[i] - vreme)
      return false;
    p = min(pozicije[i] + vreme, p + domet);
  }
  return true;
}
```

```
int main() {
  double domet;
  cin >> domet;
  int n;
  cin >> n;
  vector<double> pozicije(n);
  for (int i = 0; i < n; i++)
    cin >> pozicije[i];

  double vreme = 0;
  while (!moguceJeRasiritiVest(pozicije, domet, vreme))
    vreme += 0.001;

  cout << fixed << showpoint << setprecision(3)
       << vreme << endl;

  return 0;
}
```

Бинарна претрага

Много ефикасније, оптимално решење можемо пронаћи бинарном претрагом. Најмање могуће време је 0, највеће је једнако растојању између првог и последњег гласника $X = g_{n-1} - g_0$. Битно је нагласити монотоност, која је неопходна за бинарну претрагу (ако се порука не може доставити за неко време t, не може се доставити ни за време мање од t, а ако се може доставити за време t, може се доставити и за време веће од t). Бинарном претрагом одређујемо узак интервал $[l, d]$ такав да се порука не може пренети за l временских јединица, а може се пренети за d временских јединица и за оптимум проглашавамо средину тог интервала.

Ако се тражи прецизност од k децимала, број позива провере логаритамски зависи од производа 10^k и максималне вредности времена која је одређена растојањем између првог и последњег гласника X. Пошто се свака провера врши у времену $O(n)$, сложеност се може проценити на $O(n \log 10^k X) = O(n \cdot k \log X)$, што је прилично ефикасно.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>

using namespace std;

bool moguceJeRasiritiVest(const vector<double>& pozicije,
                          double domet, double vreme) {
  double p = pozicije[0] + vreme;
  for (int i = 1; i < pozicije.size(); i++) {
    if (p + domet < pozicije[i] - vreme)
      return false;
    p = min(pozicije[i] + vreme, p + domet);
  }
  return true;
}

int main() {
  double domet;
  cin >> domet;
  int n;
  cin >> n;
  vector<double> pozicije(n);
  for (int i = 0; i < n; i++)
    cin >> pozicije[i];
```

```
  double levo = 0, desno = pozicije[n-1] - pozicije[0];
  while (desno - levo > 0.001) {
    double s = (levo + desno) / 2.0;
    if (moguceJeRasiritiVest(pozicije, domet, s))
      desno = s;
    else
      levo = s;
  }

  cout << fixed << showpoint << setprecision(3)
       << (levo + desno) / 2.0 << endl;

  return 0;
}
```

Види другачија решења овог задатка.

2.10 Скупови и мапе (речници)

2.10.1 Скупови

У програмима често имамо потребе да одржавамо скуп елемената (без дупликата), у који ефикасно можемо да додајемо елементе, из кога ефикасно можемо да избацујемо елементе и за који ефикасно можемо да проверавамо да ли је нека задата вредност елемент скупа. Савремени програмски језици у својим библиотекама пружају структуре података које нуде баш ове операције.

У језику C++ скуп је подржан кроз две класе: `set<T>` и `unordered_set<T>`, где је `T` тип елемената скупа. Имплементација је различита (прва је заснована на балансираним бинарним дрветима, а друга на хеш таблицама), па су им временске и просторне карактеристике донекле различите.

Скупови подржавају следеће основне операције (за преглед свих операција упућујемо читаоца на документацију):

- `insert` - умеће нови елемент у скуп (ако је елемент већ у скупу, операција нема ефекта). Када се користи `set`, сложеност уметања је $O(\log k)$, где је k број елемената у скупу, а када се користи `unordered_set`, сложеност најгорег случаја је $O(k)$, док је просечна сложеност $O(1)$, при чему је амортизована сложеност узастопног додавања већег броја елемената такође $O(1)$. Нагласимо и да константе код сложености O(1) могу бити релативно велике и да уметање не можемо сматрати јако брзом операцијом. Честа операција је додавање n елемената у скуп. Најгора сложеност је ако су сви елементи различити и износи приближно $\log 1 + \log 2 + \ldots + \log n$, за шта се може показати да је $O(n \log n)$.
- `erase` - уклања дати елемент из скупа. Сложеност је иста као у случају уметања.
- `find` - проверава да ли скуп садржи дати елемент и враћа итератор на њега или `end` ако је одговор негативан. Тако се провера припадности елемента `e` скупу `s` може извршити са `if (s.find(e) != s.end()) ...` Сложеност најгорег случаја ове операције ако се користи `set` је `O(\log{k})`, а ако се користи `unordered_set` сложеност најгорег случаја је `O(k)`, али је просечна сложеност `O(1)`.
- `size` - враћа број елемената скупа

Могућа је и итерација кроз елементе скупа коришћењем петље облика `for (T element : skup)`, при чему се елементи колекције `set` набрајају у сортираном, а `unordered_set` у прилично насумичном редоследу (редослед је одређен хеш-функцијом која се користи и на њега се, као што само име `unordered` говори, не треба ослањати).

Ако се у скуп стављају елементарни типови података (бројеви, ниске, ...), тада се користе подразумевани поредак тј. хеш-функција. Ово је могуће променити, али излази ван домена овог материјала. Да би се формирао скуп елемената неког типа над којим није дефинисан подразумевани поредак нити хеш-функција (најчешће скуп структура или објеката неке класе), потребно је посебно дефинисати их.

2.10.2 Мапе (речници)

Програмски језик C++ пружа подршку за креирање мапа (речника, асоцијативних низова) који представљају колекције података у којима се кључевима неког типа придружују вредности неког типа (не обавезно истог). На пример, именима месеци (подацима типа `string`) можемо доделити број дана (податке типа `int`). Речници се представљају објектима типа `map<TipKljuca, TipVrednosti>`, дефинисаном у заглављу `<map>`. На пример,

```
map<string, int> brojDana =
{
    {"januar", 31},
    {"februar", 28},
    {"mart", 31},
    ...
};
```

Приметимо да смо иницијализацију мапе извршили тако што смо навели листу парова облика `{kljuc, vrednost}`. Иницијализацију није неопходно извршити одмах током креирања, већ је вредности могуће додавати (а и читати) коришћењем индексног приступа (помоћу заграда `[]`).

```
map<string, int> brojDana;
brojDana["januar"] = 31;
brojDana["februar"] = 28;
brojDana["mart"] = 31;
...
```

Мапу, дакле, можемо схватити и као низ тј. вектор у коме индекси нису обавезно из неког целобројног интервала облика $[0, n)$, већ могу бити произвољног типа.

Претрагу кључа можемо остварити методом `find` која враћа итератор на пронађени елемент тј. итератор иза краја мапе (који добијамо методом или функцијом `end`), ако елемент не постоји. На пример,

```
string mesec; cin >> mesec;
auto it = brojDana.find(mesec);
if (it != end(brojDana))
   cout << "Broj dana: " + *it << endl;
else
   cout << "Mesec nije korektno unet" << endl;
```

Све елементе речника могуће је исписати коришћењем петље `for`. На пример,

```
for (auto& it : brojDana)
     cout << it.first << ": " << it.second << endl;
```

Алтернативно, можемо експлицитно користити итераторе

```
for (auto it = brojDana.begin(); it != brojDana.end(); it++)
     cout << it->first << ": " << it->second << endl;
```

Итерација се врши у сортираном редоследу кључева.

Постоји и облик неуређене мапе (`unordered_map` из истоименог заглавља), која може бити у неким ситуацијама мало бржа него уређена (сортирана) мапа, но то је обично занемариво. Кључеви сортиране мапе могу бити само они типови који се могу поредити релацијским операторима, док кључеви неуређене мапе могу бити само они типови који се могу лако претворити у број (тзв. хеш-вредност). Ниске, које ћемо најчешће користити као кључеве, задовољавају оба услова.

Задатак: Дупликати

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Скупови

Библиотеке савремених програмских језика обично нуде и колекције података за репрезентовање скупова. Једна могућност је да се користи колекција заснована на балансираном бинарном дрвету. У језику C++ таква је колекција `set`. Додавање елемента у скуп се врши методом `insert` (при том се аутоматски води рачуна да се скуп не мења додавањем елемента који већ постоји), док се број елемената одређује методом `size()`.

Са имплементацијом скупа базираном на балансираном бинарном дрвету, убацивање у скуп је обично сложености $O(\log k)$, где је k број елемената у скупу, па је укупна сложеност овог приступа највише $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo elemente u skup
  set<unsigned> a;
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    unsigned x;
    cin >> x;
    a.insert(x);
  }
  // ispisujemo broj elemenata skupa
  cout << a.size() << endl;
  return 0;
}
```

Једна могућност је да се користи колекција заснована на хеширању. У језику C++ таква је колекција `unordered_set` и она се користи на исти начин као и `set` (при чему тип елемената мора да буде такав да је за њега расположива хеш-функција).

Сложеност најгорег случаја додавања у скуп заснован на хеширању је $O(k)$, где је k број елемената у скупу, али је амортизована сложеност већег броја додавања једнака $O(1)$. Зато је укупна сложеност алгоритма једнака $O(n)$ (уз релативно велики константни фактор који уметање у овакав скуп носи).

```
#include <iostream>
#include <vector>
#include <unordered_set>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo elemente u skup
  unordered_set<unsigned> a;
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    unsigned x;
    cin >> x;
    a.insert(x);
  }
  // ispisujemo broj elemenata skupa
  cout << a.size() << endl;
  return 0;
}
```

Задатак: Неупарени елемент

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Пребројавање

Један начин да се задатак реши је да се преброји колико се који број пута појављује. За сваки број треба да чувамо број његових појављивања.

Пошто је опсег бројева велики за бројање можемо користити мапу (тј. речник).

У језику C++ то може бити `map` или `unordered_map`. Након учитавања и бројања свих елемената пролазимо кроз елементе мапе, за сваки проверавамо број појављивања заустављајући се када наиђемо онај који се појављује само једном.

Ако претпоставимо да је ажурирање и читање података из мапе тј. речника сложености $O(\log k)$, где је k број елемената у мапи тј. речнику, тада је фаза пребројавања појављивања сложености $O(n \log n)$ и она доминира укупном сложеношћу решења. Пролазак кроз све елементе мапе је линеаран у односу на тај број елемената (који не може бити већи од n).

```
#include <iostream>
#include <map>

using namespace std;

int main() {
  int n;
  cin >> n;
  map<int, int> broj_pojavljivanja;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    broj_pojavljivanja[x]++;
  }
  for (auto it : broj_pojavljivanja)
    if (it.second == 1)
      cout << it.first << endl;
  return 0;
}
```

Скуп виђених елемената

Слично решење овоме је да чувамо скуп бројева виђених до тада и да у њега додамо учитани елемент ако се не налази у скупу, односно да га избацимо из скупа ако је већ раније убачен у скуп (скуп у сваком тренутку чува скуп потенцијалних кандидата за елемент који се појављује само једном - када елемент учитамо други пут он више није потенцијални кандидат и зато га избацујемо из скупа). За представљање скупа у језику C++ можемо користити `set` или `unordered_set`.

Ако претпоставимо да је уметање елемената у скуп и читање елемената скупа сложености $O(\log k)$, где је k тренутни број елемената скупа, онда је сложеност формирања скупа $O(n \log n)$. Наиме, може да се деси да се прво половина елемената убацује у скуп, па да се онда један по један елемент избацује (за избацивање је такође потребно $O(n \log n)$ корака).

```
#include <iostream>
#include <set>

using namespace std;

int main() {
```

```
  set<int> elementi;
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    auto it = elementi.find(x);
    if (it == elementi.end())
      elementi.insert(x);
    else
      elementi.erase(x);
  }
  cout << *elementi.begin() << endl;
  return 0;
}
```

Задатак: Двоструки студент

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Најбројнији елемент

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Пребројавање помоћу мапе тј. речника

У првој фази можемо пребројати колико гласова је добио сваки кандидат. За то можемо употребити асоцијативни низ (речник тј. мапу). Након тога, резултат израчунавамо као највећу вредност у мапи. Максимум можемо одредити било ручно, помоћу петље, било применом библиотечке функције `max_element`.

Ако претпоставимо да је уношење и ажурирање вредности у мапу тј. речник у којој тренутно постоји k кључева сложености $O(\log k)$, сложеност целог алгоритма је $O(n \log n)$. Наиме, након пребројавања свих елемената, највећу вредност у мапи одређујемо у линеарној сложености у односу на број кључева у њој (што је n у најгорем случају). Обратимо пажњу на то да смо у овој анализи претпоставили и да је поређење ниски које су кључеви у мапи операција константне сложености (што је оправдано јер су те ниске прилично кратке и очекивано "шаренолике", па се њихов поредак може одредити поређењем малог броја почетних карактера).

```
#include <iostream>
#include <string>
#include <map>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;

  // ucitavamo niske i racunamo broj pojavljivanja svake od njih
  map<string, int> brojPojavljivanja;
  for (int i = 0; i < n; i++) {
    string x; cin >> x;
    brojPojavljivanja[x]++;
  }

  // odredjujemo onu sa maksimalnim brojem pojavljivanja
```

```
  cout << max_element(begin(brojPojavljivanja), end(brojPojavljivanja),
                      [](auto& p1, auto& p2) {
                        return p1.second < p2.second;
                      })->second << endl;
}
```

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање уз помоћ мултискупа

Сортирање се може извршити коришћењем мултискупа. Користи се алгоритам *сортирања уметањем* (енгл. *insertion sort*), једино што уместо уметања елемената у низ врши уметање елемената у мултискуп. На крају се испишу сви елементи мултискупа редом (они се чувају у сортираном редоследу). Пошто се мултискуп имплементира коришћењем сортираног бинарног дрвета, овај се алгоритам назива и *сортирањем коришћењем дрвета* (енгл. *tree sort*).

У језику C++ можемо користити библиотечку колекцију `multiset`.

Пошто се уметање у мултиску извршава у сложености $O(\log k)$, где је k број елемената у мултискупу, а испис у времену $O(k)$, сложеност овог поступка сортирања је $O(n \log n)$. Заузеће меморије је $O(n)$, уз, додуше, мало већи константни фактор него када се елементи користе у низ. Додатно, елементи нису поређани један уз други у меморији, што мало успорава приступ.

```
int n;
cin >> n;
multiset<int> a;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  a.insert(x);
}
for (int x : a)
  cout << x << endl;
```

Види другачија решења овог задатка.

Задатак: Квадрати

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Највећи поновљени елемент

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Број различитих дужина дужи

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Број парова датог збира

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Види другачија решења овог задатка.

Задатак: Тројке датог збира (3sum)

Такмичари из програмирања имају свој рејтинг који је изражен као неки цео број. На државно екипно такмичење школе треба да пошаљу своје трочлане екипе, међутим, да би такмичење било што занимљивије, упутство организатора је да све екипе буду уједначене тј. да свака екипа на такмичењу има збирни рејтинг нула. Ако су познати рејтинзи свих такмичара из неке школе, напиши програм који одређује на колико начина школа може да одабере своју екипу.

Улаз: Са стандардног улаза се уноси број такмичара n ($3 \leq n \leq 1000$), а затим и n различитих целих бројева из интервала $[-10^6, 10^6]$, раздвојених са по једним размаком (то су рејтинзи такмичара).

Излаз: На стандардни излаз исписати број могућих трочланих екипа таквих да је укупан збир рејтинга та три члана једнак нули.

Пример

Улаз	*Излаз*
`9` `-8 -5 7 4 1 -2 9 -3 2`	`4`

Задатак: Анаграм

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Две ниске су анаграми ако се једна може добити пермутовањем редоследа карактера друге, па је централни задатак одредити да ли једна ниска представља пермутацију друге. Провера да ли је један низ пермутација другога описана је у задатку Провера пермутација и у овом задатку се могу применити све технике које су у том задатку описане. Једина разлика у односу на обичну проверу пермутација је то што се приликом провере анаграма не узимају у обзир сви карактери, већ само мала слова.

Бројање карактера

Једно решење је засновано на бројању карактера. Како анаграми настају један од другог премештањем слова јасно је да број појављивања сваког слова у првој ниски мора бити једнак броју појављивања тог слова у другој. Зато је довољно проверити да ли је број појављивања сваког слова абецеде исти и у једној и у другој ниски. Један начин је да креирамо низове од 26 елемената (26 је број слова у енглеској абецеди) који на одговарајућој позицији садрже број појављивања одговарајућег слова (на позицији 0 број појављивања слова 'a'), за сваку од ниски и да у петљи бројимо све оне карактере који су мала слова (слично као у задатку Фреквенција знака). Ниске су анаграми ако се добијени низови подударају. Ако се у језику C++ ради са обичним низовима, проверу њихове једнакости морамо вршити било помоћу петље, било библиотечком функцијом `equal`, а ако се ради са векторима, они се могу упоредити оператором `==`.

Бројање карактера је сложености $O(n)$, где је n дужина ниске (сваком бројачу у низу приступамо у времену $O(1)$), док је поређење два низа бројача једнака $O(k)$, где је k број слова абецеде. Приметимо да тај број веома мали (само 26 карактера), тако да се и меморија која се користи за низове бројача и време потребно за њихово поређење могу сматрати практично константним. Укупна временска и меморијска сложеност алгоритма је стога $O(n)$.

```
#include <iostream>
#include <string>
using namespace std;

void prebrojSlova(string s, int b[]) {
  for (char c : s)
    if (islower(c))
      b[c - 'a']++;
}

bool jednaki(int b1[], int b2[]) {
```

```
  // return equal(b1, b1 + 26, b2);
  for(int i = 0; i < 26; i++)
    if (b1[i] != b2[i])
      return false;
  return true;
}

bool anagram(string s1, string s2) {
  int b1[26] = {0}, b2[26] = {0};
  prebrojSlova(s1, b1);
  prebrojSlova(s2, b2);
  return jednaki(b1, b2);
}

int main() {
  string s1, s2;
  getline(cin, s1);
  getline(cin, s2);
  if (anagram(s1, s2))
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Задатак: Провера пермутација

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: D-пермутација

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Мултискупови елемената

Два низа се могу пермутовати један у други ако су им мултискупови елемената једнаки. Један начин да то проверимо је да експлицитно креирамо мултискупове за сваку од d партиција низа и да за два низа проверимо да ли су им одговарајуће партиције једнаке.

У језику C++ мултискупове можемо представити помоћу колекције `multiset`. Чувамо, дакле, низ (тј. вектор) од d мултискупова. Провера да ли су два мултискупа једнака се може остварити једноставно, коришћењем оператора `==`.

Ако претпоставимо да се у мутлискуп од m елемената нови елемент може додати у времену $O(\log m)$, тада је за формирање d мултискупова са по неких k/d елемената потребно време $O(d\frac{k}{d}\log\frac{k}{d}) = O(k\log\frac{k}{d})$. Пошто се ово израчунава за сваки од n низова, време потребно за канонизацију је $O(n \cdot k\log\frac{k}{d})$. Пошто се провера једнакости два мутлискупа од по k елемената може извршити у времену $O(k)$, а провера се врши $n-1$ пут, након канонизације врши се још $O(n\cdot k)$ операција. Сложеност зато можемо проценити на $O(n\cdot k\cdot\log(k/d))$, тј. $O(k\cdot\log(k/d))$ јер n, за разлику од k има веома малу вредност (практично се може сматрати константом).

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;
```

```
vector<multiset<int>> prebroj(const vector<int>& s, int d)
{
  int duzina = s.size();
  // pomocni vektor koji ce cuvati particiju po particiju
  vector<multiset<int>> particije(d);

  // kanonizovacemo svaku od d particija
  for (int i = 0; i < d; i++)
    for (int j = i; j < duzina; j += d)
      particije[i].insert(s[j]);


  return particije;
}

int main() {
  // iskljucujemo sinhronizaciju da bi ucitavanje teklo brze
  ios::sync_with_stdio(false);

  // ucitavamo rastojanje d
  int d;
  cin >> d;
  // ucitavamo dužinu nizova k
  int k;
  cin >> k;
  // ucitavamo broj nizova
  int n;
  cin >> n;

  // ucitavamo niz S
  vector<int> s(k);
  for (int i = 0; i < k; i++)
    cin >> s[i];

  // odredjujemo kanonskog predstavnika niza s
  auto s_particije = prebroj(s, d);

  // obradjujemo preostalih n-1 nizova Ti
  for (int j = 1; j < n; j++) {
    // ucitvamo niz Ti
    vector<int> t(k);
    for (int i = 0; i < k; i++)
      cin >> t[i];

    // odredjujemo kanonskog predstavnika niza Ti
    auto t_particije = prebroj(t, d);

    // odredjujemo da li se niz S moze transformisati u niz Ti
    // (moze ako i samo ako su im kanonski predstavnici jednaki)
    // i ispisujemo rezultat u trazenom formatu
    cout << (s_particije == t_particije ? "da" : "ne") << endl;
  }

  return 0;
}
```

Задатак: Анаграми

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

О техникама провере да ли су две речи анаграми било је речи у задатку Анаграм. Провера се може заснивати било на сортирању слова две речи и затим поређењу да ли су добијене речи исте, било на поређењу броја појављивања сваког слова. Дакле, сваку реч можемо нормализовати или тако што је сортирамо или тако што израчунамо број појављивања сваког од 26 карактера енглеске абецеде. Тиме добијамо низ канонских представника сваке речи и желимо да у том низу пронађемо скуп еквивалентних елемената (једнаких канонских представника) који је највеће кардиналности од свих таквих подскупова. То је веома слично задатку Фреквенције речи и опет се може решити било сортирањем, па тражењем најдуже серије једнаких узастопних елемената (као у задатку Најдужа серија победа или Најдужа растућа серија), било бројањем појављивања сваког канонског представника.

Бројање канонских представника

Једно могуће решење је направимо мапу тј. речник који канонске представнике (било да су то сортиране речи, било низови бројача карактера) пресликавају у њихов број појављивања (тада не вршимо сортирање низа низова). У језику C++ вектори могу да буду кључеви мапе (јер је на њима дефинисано лексикорафско поређење помоћу оператора <).

Формирање канонских представника захтева $O(n \cdot m \log m)$ корака ако се слова речи сортирају или $O(n \cdot m)$ корака ако се карактери броје. Пошто је m релативно мали број у односу на n и можемо га сматрати константним, ово у оба случаја грубо можемо оценити као $O(n)$. Ако претпоставимо да убацивање елемента у мапу тј. речник која садржи k елемената захтева $O(\log k)$ корака, тада формирање мапе од n речи захтева $O(n \log n)$ корака (овде је претпостављено да се однос два кључа може одредити практично у константној сложености), што доминира сложеношћу целог алгоритма.

```
#include <iostream>
#include <vector>
#include <string>
#include <map>

using namespace std;

vector<int> prebrojSlova(const string& s) {
  vector<int> rez(26);
  for (char c : s)
    rez[c - 'a']++;
  return rez;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  map<vector<int>, int> broj_reci;
  for (int i = 0; i < n; i++) {
    string s;
    cin >> s;
    broj_reci[prebrojSlova(s)]++;
  }

  int max = 0;
  for (auto it : broj_reci)
    if (it.second > max)
      max = it.second;
  cout << max << endl;
}
```

2.11 Инкременталност

Једна од основних техника за избегавање лоше сложености алгоритама је да се избегне израчунавање истих ствари више пута у истом програму. Често је потребно израчунати неку вредност за различите вредности неког параметра. Рећи ћемо да је израчунавање **инкрементално** ако се резултујућа вредност за наредну вредност параметра израчунава коришћењем већ израчунатих вредности за претходну (или неколико претходних) вредности параметра. Дакле, на мале измене улазних података реагујемо малим изменама резултата (уместо да поново вршимо велико израчунавање свега, из почетка).

Веома једноставан пример принципа инкременталности је израчунавање парцијалних збирова (збирови префикса) елемената неког низа. На пример, ако је дат низ $1, 2, 3, 4, 5$, његови парцијални збирови су редом $0, 1, 3, 6, 10, 15$. Веома једноставно се примећује да се израчунавање наредног парцијалног збира не мора вршити сабирањем свих елемената од почетка, већ се може добити сабирањем претходног парцијалног збира са текућим елементом низа (на пример, збир $1 + 2 + 3 + 4 = 10$, се добија сабирањем претходног збира $1 + 2 + 3 = 6$ и текућег елемента 4). Ако збир првих k елемената означимо са Z_k, тада важи да је $Z_0 = 0$ и да је $Z_{k+1} = Z_k + a_k$. Овим смо добили серију бројева у којој се наредни елемент израчунава на основу претходног (или неколико претходних). За такве серије кажемо да су *рекурентне серије*. Сваки наредни члан се израчунава у сложености $O(1)$, па се израчунавање свих парцијалних збирова низа дужине n врши у сложености $O(n)$. Када би се сваки парцијални збир рачунао сабирањем елемената низа из почетка, тада би израчунавање k-тог збира било сложености $O(k)$, а израчунавање свих збирова сложености $O(n^2)$.

Принцип инкременталности је у тесној вези са индуктивно/рекурзивном конструкцијом алгоритама и лежи у основи великог броја основних алгоритама. Већ само израчунавање збира свих елемената низа заправо почива на постепеном, инкременталном израчунавању збирова префикса, све док се не израчуна збир свих елемената низа. Слично је и са израчунавањем минумума, максимума, линеарном претрагом и другим фундаменталним алгоритмима. У свим овим примерима крећемо од неке почетне вредности у низу резултата, а затим наредну вредност у том низу израчунавамо на основу претходне или неколико претходних, што директно одговара индуктивном поступку израчунавања. Слична техника (добијања наредних резултата на основу претходних) примењује се у склопу технике динамичког програмирања навише, о чему ће више речи бити касније.

Поред парцијалних збирова, инкрементално се могу израчунавати и парцијални производи, парцијални минимуми и максимуми и слично.

2.11.1 Инкременталност збира и производа

Једна од најчешће коришћених статистика је збир елемената неког низа. Парцијални збирови серије елемената се могу рачунати инкрементално, што често убрзава алгоритам. Веома сличан збиру је и производ, и парцијалне производе је такође могуће рачунати инкрементално.

Задатак: Префикс највећег збира

Сваког дана током неког периода на банковни рачун је вршена тачно једна трансакција (уплата или исплата новца). Ако је почетно стање на рачуну нула, напиши програм који одређује највеће стање на рачуну током тог периода.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 100000$), а затим и n целих бројева (сваки у посебној линији) који представљају трансакције (позитиван број означава уплату, а негативан исплату).

Излаз: На стандардни излаз исписати један цео број који представља највеће стање на рачуну у неком тренутку.

Пример

Улаз	*Излаз*
5 4 2 -3 5 -4	8

Решење

Засебно израчунавање сваког салда

Под претпоставком да су подаци о свим трансакцијама учитане у низ, директно, наивно решење овог задатка би било да се за сваки дан k (од 0 до $n-1$) одреди збир Z_k елемената од првог дана закључно са даном k (алгоритмом сабирања серије бројева) и да се онда одреди највећи од тих збирова (алгоритмом одређивања минимума тј. максимума описаним).

Пошто се збир префикса дужине k може израчунати у k корака, укупно бисмо извршавали око $1+2+\ldots+n = \frac{n(n+1)}{2}$ сабирања (и линеарни број ажурирања максимума) и временска сложеност овог наивног алгоритма била би квадратна у односу на број елемената низа тј. $O(n^2)$.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ubrzavamo ucitavanje
  ios_base::sync_with_stdio(false);
  // ucitavamo sve elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // prvog dana je zbir 0
  // najveci zbir prefiksa
  int maxZbir = 0;
  for (int i = 0; i < n; i++) {
    // racunamo zbir prefiksa od pocetka do pozicije i (ukljucujuci i nju)
    int zbir = 0;
    for (int j = 0; j <= i; j++)
      zbir += a[j];
    // azuriramo maksimum ako je potrebno
    if (zbir >= maxZbir)
      maxZbir = zbir;
  }

  // ispisujemo maksimum
  cout << maxZbir << endl;
  return 0;
}
```

Скривена сложеност

За сабирање је могуће употребити и библиотечку функцију.

Иако се тада у програму види једна петља, због скривене линеарне сложености библиотечких функција укупна сложеност програма остаје квадратна тј. $O(n^2)$.

```
#include <iostream>
#include <vector>
#include <numeric>

using namespace std;

int main() {
  // ubrzavamo ucitavanje
  ios_base::sync_with_stdio(false);
```

```
  // ucitavamo sve elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // prvog dana je zbir 0
  // najveci zbir prefiksa
  int maxZbir = 0;
  for (int i = 0; i < n; i++) {
    // racunamo zbir prefiksa od pocetka do pozicije i (ukljucujuci i nju)
    int zbir = accumulate(begin(a), next(begin(a), i+1), 0);
    // azuriramo maksimum ako je potrebno
    if (zbir >= maxZbir)
      maxZbir = zbir;
  }

  // ispisujemo maksimum
  cout << maxZbir << endl;
  return 0;
}
```

Инкрементално израчунавање салда

Много ефикасније решење се може добити ако се примети прилично очигледна чињеница да се салдо у следећем дану може јако једноставно израчунати ако је познат салдо у претходном дану тако што се тај салдо увећа за износ трансакције у следећем дану, што омогућава *инкременталност* израчунавања. Збир (текући салдо) израчунавамо индуктивном конструкцијом. Означимо са Z_k збир трансакција у првих k дана. Важи да је $Z_0 = 0$ и да је $Z_{k+1} = Z_k + a_k$. Збир зато иницијализујемо на нулу, а онда у петљи учитавамо трансакцију по трансакцију, увећавамо тренутни збир за износ трансакције, проверавамо да ли је текући износ већи од до тада највећег и ако јесте, ажурирамо вредност највећег износа.

У сваком кораку петље вршимо константан број додатних операција (ажурирање збира и минимума) и сложеност овог алгоритма је линеарна тј. $O(n)$.

Приметимо и да је меморијска сложеност овог решења $O(1)$, јер током инкременталне обраде нема потребе памтити вредности свих трансакција у низу.

```
#include <iostream>

using namespace std;

int main() {
  // ubrzavamo ucitavanje
  ios_base::sync_with_stdio(false);
  // ucitavamo broj elemenata niza
  int n;
  cin >> n;

  // prvog dana je saldo 0
  // najveci zbir prefiksa
  int maxZbir = 0;
  // tekuci zbir prefiksa
  int zbir = 0;
  for (int i = 0; i < n; i++) {
    // ucitavamo tekuci element niza
    int x; cin >> x;
    // azuriramo zbir prefiksa
    zbir += x;
    // azuriramo maksimum ako je potrebno
```

```
    if (zbir >= maxZbir)
      maxZbir = zbir;
  }

  // ispisujemo resenje
  cout << maxZbir << endl;
  return 0;
}
```

Задатак: Сегмент датог збира у низу целих бројева

Напиши програм који за дати низ целих бројева одређује број непразних сегмената узастопних елемената низа чији је збир једнак датом броју.

Улаз: Са стандардног улаза се у првој линији уноси тражена вредност збира z (цео број -10000 и 10000), затим, у наредној линији димензија низа n ($3 \leq n \leq 50000$) и затим у n наредних линија елементи низа (цели бројеви између -100 и 100).

Излаз: На стандардни излаз испиши број сегмената чији је збир једнак z.

Пример

Улаз	*Излаз*
11 10 1 2 3 5 1 -1 1 5 3 2	7

Решење

Груба сила

Директно решење грубом силом подразумевало би да се провере сви сегменти узастопних елемената, да се за сваки израчуна збир и да се провери да ли је тај збир једнак траженом. Сви сегменти се могу набројати угнежђеним петљама, где спољна петља пролази кроз леве крајеве сегмента (i узима вредности од 0 па до $n-1$), а унутрашња петља пролази кроз десне крајеве сегмената (од вредности i па до $n-1$). Сличну технику смо видели, на пример, у задатку . Збир можемо рачунати алгоритмом сабирања или применом библиотечких функција. Сличну технику смо видели, на пример, у задатку Збир n бројева.

Сложеност овог приступа је кубна у односу на димензију n тј. $O(n^3)$. Наиме, постоји квадратни број сегмената и за сабирање елемената сваког од њих потребно је линеарно време.

Претходна процена је извршена прилично грубо, јер иако је сложеност сабирања сегмента линеарна, нису сви сегменти дужине n, међутим, и прецизнија анализа ће довести до истог резултата. Унутрашња петља се извршава $j - i + 1$ пута толико пута се врши операција сабирања), што одговара дужини сегмента. Спољне петље које набрајају све сегменте набрајају један сегмент дужине n, два сегмента дужине $n-1$, три сегмента дужине $n-2$, … и n сегмената дужине 1. Значи да је број корака једнак $1 \cdot n + 2 \cdot (n-1) + 3 \cdot (n-2) + \ldots + (n-1) \cdot 2 + n \cdot 1$. Дакле, сегмената дужине k има $n - k + 1$, па важи да је претходни збир једнак

$$\sum_{k=1}^{n} k(n-k+1) = (n+1) \cdot \sum_{k=1}^{n} k - \sum_{k=1}^{n} k^2 = (n+1) \cdot \frac{n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6}$$

Ово је реда величине $\frac{n^3}{2} - \frac{2n^3}{6} = \frac{n^3}{6}$, што је $O(n^3)$.

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>
#include <numeric>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  // ucitavamo trazeni zbir
  int trazeniZbir;
  cin >> trazeniZbir;

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // broj segmenata sa trazenim zbirom
  int broj = 0;

  // prolazimo sve segmente
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      // izracunavamo zbir segmenta [i, j]
      int zbir = accumulate(next(a.begin(), i), next(a.begin(), j+1), 0);
      // proveravamo da li je zbir tog segmenta jednak trazenom
      if (zbir == trazeniZbir)
        broj++;
    }
  }

  cout << broj << endl;

  return 0;
}
```

Инкрементално рачунање збира сегмената

Претходни алгоритам се може унапредити ако се збирови рачунају инкрементално тј. ако се искористи чињеница да се збир сваког сегмента који се добија проширивањем претходног сегмента једним елементом може лако израчунати на основу збира претходног сегмента, тако што се збир претходног сегмента увећа за текући елемент низа. Ту идеју смо видели, на пример, у задатку Префикс највећег збира.

Дакле, опет можемо набрајати све сегменте угнежђеним петљама, на почетку тела спољашње петље збир иницијализујемо на нулу, у унутрашњој петљи збир увећавамо за елемент a_j и, ако је он једнак траженом, исписујемо индексе интервала $[i, j]$.

Овим је избегнута линеарна сложеност за израчунавање збира тренутног интервала тј. да се збир сваког наредног интервала добија у константном времену, тако да је сложеност целог алгоритма редукована на квадратну тј. $O(n^2)$.

Опет би се прецизнијом анализом добио исти резултат. Наиме, сви збирови сегмената који почињу на позицији 0 рачунају се помоћу n сабирања, збирови сегмената који почињу на позицији 1 рачунају се помоћу $n-1$ сабирања итд. Укупан број сабирања је, дакле, $1 + \ldots + n = \frac{n(n+1)}{2}$, што је $O(n^2)$.

Задатак се, може решити и доста ефикасније од овога.

```cpp
#include <iostream>
#include <vector>
```

```cpp
using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  // ucitavamo trazeni zbir
  int trazeniZbir;
  cin >> trazeniZbir;

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int broj = 0;

  // prolazimo sve segmente
  for (int i = 0; i < n; i++) {
    // zbir segmenta [i, j]
    int zbir = 0;
    for (int j = i; j < n; j++) {
      // izracunavamo zbir segmenta [i, j] na osnovu zbira segmenta [i, j-1]
      zbir += a[j];
      // proveravamo da li je zbir tog segmenta jednak trazenom
      if (zbir == trazeniZbir)
        broj++;
    }
  }

  cout << broj << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Факторијели од 1 до n

Написати програм којим се исписују вредности факторијела свих природних бројева од 1 до n. Факторијел броја k у ознаци $k!$ је производ природних бројева од 1 до k ($k! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot k$).

Улаз: Прва линија стандарног улаза садржи природан број $n \leq 20$.

Излаз: Стандардни излаз садржи редом факторијеле природних бројева од 1 до n сваки факторијел у посебној линији.

Пример

Улаз	*Излаз*
4	1
	2
	6
	24

Решење

За сваки број од 1 до n треба исписати његов факторијел. Задатак решавамо тако за сваки број k који узима вредности од 1 до n рачунамо факторијел као производ $1 \cdot 2 \cdot 3 \cdot \cdots \cdot k$.

Функција за израчунавање сваког факторијела засебно

Једно решење је, да за сваки број k рачунамо факторијел на уобичајан начин, како је описано у задатку

Факторијел множимо бројеве полазећи од 1 до k. Израчунавање факторијела можемо реализовати у посебној функцији, што је у складу са добром праксом организовања кода, али је, нажалост, у овом контексту неефикасно.

У кораку k вршимо $k-1$ множења, па укупно вршимо $1+2+\ldots+(n-1)$ тј. $\frac{n(n-1)}{2}$ множења, тако да је овај алгоритам квадратне сложености $O(n^2)$.

```
#include <iostream>

using namespace std;

unsigned long long faktorijel(int k) {
  unsigned long long p = 1;
  for (int i = 2; i <= k; i++)
    p *= i;
  return p;
}

int main() {
  int n;
  cin >> n;
  for(int k = 1; k <= n; k++)
    cout << faktorijel(k) << endl;
  return 0;
}
```

Инкрементално израчунавање серије факторијела

Приметимо да је израчунавање свих потребних факторијела заправо израчунавање парцијалних производа серије бројева $1, 2, \ldots, n$. Слично као што смо у задатку Префикс највећег збира, све парцијалне збирове рачунали инкрементално, тако у овом задатку све парцијалне производе можемо рачунати инкрементално.

Приликом израчунавања факторијела броја k, ми заправо морамо израчунати и факторијеле свих бројева мањих од k. Тако на пример приликом рачунања $5!$ треба да израчунамо $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$, а да бисмо то израчунали потребно је да у неком тренутку знамо вредност производа $1 \cdot 2 \cdot 3 \cdot 4$ тј. $4!$. Можемо уочити да је $0! = 1$ и да за $k \geq 0$ важи $(k+1)! = k! \cdot (k+1)$. Међутим, $k!$ је управо вредност коју смо рачунали у претходном кораку и заправо је довољно само претходни производ помножити са 5. Ако са P_k обележимо производ првих k природних бројева тј. $k!$, важи да је $P_0 = 1$ и $P_{k+1} = P_k \cdot (k+1)$, чиме добијамо рекурентну серију. Да бисмо имплементирали инкрементално решење потребно је памтити претходно израчунати факторијел. То реализујемо коришћењем променљиве p. На почетку поставимо p на вредност 1. Затим редом за сваки број k од 1 до n, претходни факторијел помножимо са k (p добија вредност $p \cdot k$) прикажемо нову вредност факторијела, и наставимо петљу даље.

Инкременталним решењем вршимо по једно множење у n корака петље, што даје алгоритам линеарне сложености $O(n)$. Пошто је факторијел веома брзо растућа функција у овом задатку се разлика не осећа на тест-примерима (вредност n је веома мала), међутим, у наредним задацима видећемо како инкрементални приступ може значајно да унапреди ефикасност решења.

```
#include <iostream>

using namespace std;

int main() {
  int n; cin >> n;
  unsigned long long p = 1;
  for(int k = 1; k <= n; k++) {
    p *= k;
    cout << p << endl;
  }
  return 0;
}
```

Задатак: Хармонијски пи

Збир

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots$$

једнак је $\frac{\pi}{4}$. Напиши програм који за унето n процењује вредност броја π израчунавајући првих n сабирака претходног збира, а затим показује како се вредност мења променом броја сабирака тако што програм приказује вредности збира за сваки број сабирака од 1 до n. Рецимо и да је ово веома лош начин за процену броја π, јер је, како ћете видети, потребно узети велики број сабирака да би се приближили стварној вредности броја π.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 50000$).

Излаз: На стандардни излаз прво исписати збир израчунат са n сабирака, заокружен на 5 децимала, а затим и остале тражене збирове (све заокружене на 5 децимала), сваки у посебном реду.

Пример

Улаз	*Излаз*

`5`	`3.33968`
	`4.00000`
	`2.66667`
	`3.46667`
	`2.89524`
	`3.33968`

Задатак: Сума реда

Другарице су кренуле на клизање. Изнајмиле су клизаљке, мало се клизале, а онда одлучиле да се окрепе уз топли чај. Изуле су се, ставиле клизаљке на гомилу, али су заборавиле да обележе које клизаљке су чије. Пошто све имају ногу веома сличне величине, договориле су се да није ни важно, него да свака може да узме било који пар клизаљки са гомиле. Израчунати колика је вероватноћа да ниједна од њих није добила исте клизаљке које је користила пре паузе за чај, ако се зна да се та вероватноћа може израчунати као $1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} \dots + \frac{(-1)^n}{n!}$, где је n број другарица.

Улаз: Са стандардног улаза се учитава број другарица n ($2 \leq n \leq 20$).

Излаз: На стандардни излаз исписати тражену вероватноћу, заокружену на 14 децимала.

Пример 1

Улаз	*Излаз*
`2`	`0.500000000000000`

Пример 2

Улаз	*Излаз*
`10`	`0.367879464285714`

Задатак: Оптимални сервис

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Види другачија решења овог задатка.

Задатак: Најкраћи сегмент збира бар K

Једна финансијска компанија тргује на берзи током n дана. Позната је њена зарада током сваког дана (она може бити и негативна, ако је комапнија тог дана трговала неповољно). Одредити најмањи број узастопних дана у ком је збир зарада компаније бар k.

Улаз: Са стандардног улаза се уноси цео број k ($1 \leq k \leq 10^5$), а затим број дана n ($1 \leq n \leq 10^5$) и у наредном реду зарада током n дана (цели бројеви између и -2000 и 2000).

Излаз: На стандардни излаз исписати тражени најмањи број дана или - ако компанија током тих n дана никада није остварила збирну зараду k.

Пример 1

Улаз	*Излаз*
15 10 1 -2 3 4 5 4 3 2 -1 2	4

Објашњење

Укупна зарада 15 се, на пример, остварује током дана у којима је зарада 3, 4, 5, 4.

Пример 2

Улаз

```
13
10
1 -2 3 4 5 4 3 2 -1 2
```

Објашњење

Укупна зарада 13 се, на пример, остварује током дана у којима је зарада 4, 5, 4.

Излаз

```
3
```

Пример 3

Улаз

```
30
10
1 -2 3 4 5 4 3 2 -1 2
```

Излаз

```
-
```

Решење

Груба сила уз инкременталност и одсецање

Решење грубом силом подразумева да се израчуна збир сваког сегмента и упореди са K. Угнежђеним петљама анализирамо све сегменте (спољном леви, а унутрашњом десни крај), а збир сегмента рачунамо инкрементално (слично као у задатку Префикс највећег збира). Израчунавање можемо убрзати коришћењем две врсте одсецања.

Чим се пронађе први сегмент $[i, j]$ чији је збир већи или једнак од K, можемо престати са анализом сегмената који почињу на позицији i, јер ако постоји неки други сегмент $[i, j']$ чији је збир већи или једнак од K, он ће сигурно бити дужи од сегмента $[i, j]$. Дакле, чим наиђемо на први сегмент $[i, j]$ чији је збир бар K, упоређујемо његову дужину $j - i + 1$ са дужином најкраћег до тада пронађеног сегмента и ту дужину ажурирамо ако је потребно. Минималну дужину можемо иницијализовати на вредност $n + 1$ (ако остане $n + 1$, знамо да не постоји ни један сегмент чији је збир бар K, јер чим се пронађе први такав сегмент, вредност минимума ће пасти на вредност која је мања или једнака од n).

Друго одсецање се заснива на томе да анализу свих сегмената $[i, j]$ који почињу на позицији i можемо прекинути чим дужина $j-i+1$ достигне тренутну минималну дужину сегмента, јер ако се и пронађу неки сегменти чији је збир бар K, они сигурно неће бити мање дужине од минималне и неће допринети њеном смањењу.

Иако одсецања допреносе ефикасности алгоритама у великом броју примера (нарочито ако се рано у претрази пронађе неки кратак сегмент чији је збир бар K), сложеност најгорег случаја је $O(n^2)$ (она наступа, на пример, када је K свих елемената позитивног низа, тј. када не постји ни један сегмент чији је збир бар K).

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // potrebni zbir
```

```
  int k;
  cin >> k;

  // ucitavamo sve elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // duzina najkraceg segmenata (+beskonacno na pocetku)
  int minBrojElemenata = n + 1;
  // analiziramo sve leve krajeve segmenta
  for (int i = 0; i < n; i++) {
    // zbir segmenta [i, j]
    int zbir = 0;
    // prosirujemo segment [i, j] sve dok mu zbir ne postane bar K
    int j;
    for (j = i; j < n && j - i + 1 < minBrojElemenata; j++) {
      zbir += a[j];
      if (zbir >= k)
        break;
    }
    // ako je zbir segmenta [i, j] bar K, azuiriramo duzinu
    // najkraceg segmenta u odnosu na duzinu segmenta [i, j]
    if (zbir >= k)
      minBrojElemenata = min(minBrojElemenata, j - i + 1);
  }

  // ispisujemo resenje
  if (minBrojElemenata <= n)
    cout << minBrojElemenata << endl;
  else
    cout << "-" << endl;

  return 0;
}
```

Види другачија решења овог задатка.

2.11.2 Инкременталност минимума и максимума

И минимум и максимум серије бројева представљају веома важне и често коришћене статистике. Минимум тј. максимум серије префикса тј. серије суфикса низа такође могу да се израчунавају инкрементално, што доводи до ефикаснијих решења.

Задатак: Најбољи "сабмит"

Такмичар је током интергалактичког шампионата у програмирању слао на оцењивање један задатак више пута. Број поена које такмичар добија за задатак се рачуна тако што се одреди највећи број поена од свих појединачних слања (гледа се "најбољи сабмит"). Напиши програм који одређује колико је поена за тај задатак ученик имао након сваког слања. Пошто су интергалактички задаци веома тешки, они носе пуно поена и такмичари их често шаљу велики број пута.

Улаз: У првој линији стандардног улаза налази се природан број n ($n \leq 50000$). У следећих n линија налазе се редом поени које је ученик добио за свако појединачно слање (број између 0 и 100000).

Излаз: На стандардном излазу приказати n линија које приказују колико је поена за тај задатак ученик имао након сваког слања.

Пример

Улаз	*Излаз*
5	3
3	3
2	4
4	4
1	5
5	

Решење

За сваку позицију i у серији бројева је потребно одредити највећи међу свим бројевима од почетка серије, закључно са тренутним елементом $m_i = \max(a_0, \ldots, a_i)$.

Груба сила

Наивни начин да се то уради је да се резултати свих слања упишу у низ, а затим да се за сваку позицију i од 0 до $n-1$ одреди и испише максимум m_i. Одређивање максимума се може урадити уобичајеним алгоритмом одређивања максимума серије бројева. Тај алгоритам је приказан у задатку Најнижа температура.

Ово решење је прилично неефикасно (има квадратну сложеност у односу на број слања, тј. сложеност $O(n^2)$).

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> poeni(n);
  for (int i = 0; i < n; i++)
    cin >> poeni[i];

  for (int i = 0; i < n; i++) {
    int maxPoena = poeni[0];
    for (int j = 1; j <= i; j++)
      if (poeni[j] > maxPoena)
        maxPoena = poeni[j];

    cout << maxPoena << endl;
  }
  return 0;
}
```

Библиотечке функције - скривена сложеност

Максимум дела низа можемо одредити и библиотечком функцијом. У језику C++ можемо употребити функцију `max_element`.

Иако сада програм има само једну петљу, у функцији за израчунавање максимума је скривена линеарна сложеност, па укупна је сложеност $O(n^2)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
```

```
  vector<int> poeni(n);
  for (int i = 0; i < n; i++)
    cin >> poeni[i];

  for (int i = 1; i <= n; i++)
    cout << *max_element(begin(poeni), next(begin(poeni), i)) << endl;

  return 0;
}
```

Инкременталност серије парцијалних максимума

Кључни увид којим се може доћи до ефикаснијег решења је да се приликом додавања новог елемента максимум проширене серије m_{i+1} може израчунати крајње једноставно ако је познат максимум m_i серије пре проширења. Наиме, ако је познат број поена такмичара пре текућег слања задатка, када се одреди број поена у том слању, максимум се или увећава (ако је у текућем слању остварен највећи број поена до тада) или се не мења тј. $m_{i+1} = \max(m_i, a_{i+1})$. Почетни максимум може бити постављен или на први елемент низа (важи да је $m_0 = a_0$), или, пошто су сви елементи ненегативни, на нулу (важи да је $m_{-1} = 0$). Дакле, принцип инкременталности који важи за парцијалне збирове, важи и за парцијалне максимуме. Инкрементално израчунавање серије парцијалних збирова (збирова префикса) описано је, на пример, у задатку Префикс највећег збира.

Дакле, у задатку се примењује класичан алгоритам одређивања максимума серије бројева, али се у сваком кораку исписује текућа вредност максимума. Алгоритам за одређивање максимума серије елемената приказан приказан је у задатку Најнижа температура.

Иницијализација се врши у времену $O(1)$ и сваки нови максимум се од претходног добија у времену $O(1)$, па се n максимума израчунава у укупном времену $O(n)$.

```
#include <iostream>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  int maxPoena = 0;
  for (int i = 0; i < n; i++) {
    int poeni;
    cin >> poeni;
    if (poeni > maxPoena)
      maxPoena = poeni;
    cout << maxPoena << '\n';
  }
  return 0;
}
```

Задатак: Поглед на реку

У једној улици која иде ка реци налазе се разне куће и зграде. Инвеститор бира локацију на којој би изградио вишеспратницу, такву да се са њеног последњег спрата види река. Зато она мора бити виша или бар једнаке висине од свих постојећих зграда од одабране локације до краја улице. Напиши програм који за сваку локацију у тој улици одређује минималну висину нове вишеспратнице.

Улаз: У првој линији стандардног улаза налази се природан број n ($n \leq 50000$). У следећих n линија налазе се редом висине свих зграда и кућа од почетка до краја улице.

Излаз: На стандардном излазу приказати n бројева (сваки у посебном реду) који приказују тражене висине.

Пример

Улаз	*Излаз*
5 13 23 11 17 13	23 23 17 17 13

Задатак: Ред особа

У реду испред шалтера за продају карти за фудбалску утакмицу налази се n особа. Познате су висине особа које стоје у реду, и то редом од прве до последње особе у реду. Написати програм којим се одређује колико особа из реда службеник на шалтеру може да види, ако он види оне особе у реду испред којих су све ниже особе.

Улаз: Прва линија стандарног улаза садржи природан број n ($5 \leq n \leq 50000$) који представља број особа које се налазе у реду. Свака од наредних n линија садржи по један природан број, ти бројеви представљају висине редом од прве до последње особе у реду (висине могу бити изражене у произвољним јединицама мере, па могу бити бројеви између 10 и 10^5).

Излаз: У једној линији стандардног излаза приказати број особа из реда које службеник може да види.

Пример

Улаз	*Излаз*
8 165 178 170 178 183 176 168 183	3

Задатак: Најдужи сегмент који садржи узастопне бројеве

У низу различитих целих бројева наћи дужину најдужег сегмента (који чине узастопни елементи низа) који садржи бројеве који се могу уредити у низ узастопних целих бројева.

Улаз: У првој линији стандардног улаза уноси се број елемената низа n ($1 \leq n \leq 10000$), а затим у следећих n линија налазе се редом елементи низа.

Излаз: На стандардном излазу приказати дужину најдужег сегмента чији се елементи могу уредити у низ узастопних целих бројева.

Пример

Улаз	*Излаз*
11 55 56 58 57 90 92 94 93 91 59 60	5

Решење

Овај задатак је сличан задатку Најбројнији подскуп који садржи узастопне целе бројеве, али је решење потпуно другачије, јер није могуће елементе узимати из целог низа, већ само из сегмената узастопних елемената.

Основно питање, је како за дати сегмент испитати да ли се бројеви у њему могу уредити тако да сачињавају низ узастопних целих бројева. Један начин је да се сегмент сортира, међутим, захваљујући чињеници да су сви бројеви у низу различити, до решења можемо доћи и ефикасније. На основу задатка Да ли серија садржи узастопне природне бројеве знамо да се низ различитих бројева може уредити у низ узастопних бројева ако и само ако је $max - min + 1 = n$, где су max и min најмањи и највећи елемент, а n дужина низа. По истом принципу знамо да је потребан и довољан услов да се елементи из сегмента одређеног позицијама $[i, j]$ могу уредити у узастопан низ целих бројева је да је $max - min + 1 = -i + 1$ (где су max и min највећи и најмањи елемент унутар тог сегмента, а $j - i + 1$ је његова дужина).

Груба сила - провера свих сегмената

Наивни приступ решавању овог проблема захтевао би анализу свих сегмената и проверу за сваки сегмент да ли се у њему налазе узастопни цели бројеви из неког интервала. Међу свим сегментима у којима је то случај, потребно је пронаћи најдужи. На основу претходног критеријума за сваки сегмент потребно је одредити минимални и максимални елемент сегмента, проверити да ли је $max - min + 1$ једнака дужини сегмента и, ако јесте, проверити да ли је дужина сегмента већа од дужине за сада најдужег сегмента који испуњава услов и у складу са тим кориговати вредност дужине најдужег сегмента.

Пошто постоји $O(n^2)$ сегмената и пошто је сложеност израчунавања минимума и максимума сваког од њих линеарна у односу на дужину сегмената, може се доказати да је сложеност овог приступа је $O(n^3)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // najveca duzina segmenta
  int maxDuzina = 0;

  // razmatramo sve segmente [i, j], za i <= j
  for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++) {
      // razmatramo segment [i, j]

      // duzina segmenta
      int duzina = j - i + 1;

      // odredjujemo mu minimum i maksimum
      int max = *max_element(next(a.begin(), i), next(a.begin(), j+1));
      int min = *min_element(next(a.begin(), i), next(a.begin(), j+1));

      int raspon = max - min + 1;
      if (raspon == duzina) {
        // ako je potrebno, azuriramo maksimalnu duzinu takvog
        // segmenta
        if (duzina > maxDuzina)
          maxDuzina = duzina;
      }
```

```
    }

  // prijavljujemo rezultat
  cout << maxDuzina << endl;
  return 0;
}
```

Инкрементално израчунавање минимума и максимума растућих сегмената

Приметимо да поступак налажења максималног и минималног елемента сегмента можемо убрзати. Ако редом анализирамо све сегменте који почињу на некој позицији i, добијамо растући низ сегмената. У задацима Најбољи "сабмит" и Ред особа видели смо да се максимум и минимум сваког следећег сегмента могу једноставно одредити ако су познати максимум и минимум претходног сегмента тј. да се серија максимума и минимума може рачунати инкрементално. Наиме, суседни се сегменти при оваквој анализи разликују у једном елементу. Сваки сегмент има један елемент више него претходни, па при одређивању минималног и максималног елемента сегмента, можемо искористити вредност максималног и минималног елемента претходно анализираног сегмента. Вредност макисмалног и минималног елемента претходног сегмента поредимо са последњим елементом текућег сегмента и по потреби коригујемо вредност макисмалног и минималог елемента текућег сегмента.

Опет анализирамо $O(n^2)$ различитих сегмената, међутим, овај пут за сваки извршавамо само константан број операција, па је сложеност овог приступа $O(n^2)$.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // najveca duzina segmenta
  int maxDuzina = 0;
  // svi moguci levi krajevi segmenta
  for (int i = 0; i < n; i++) {
    // maksimum i minimum segmenta [i, i]
    int max = a[i], min = a[i];

    // razmatramo sve segmente [i, j] za i <= j
    for (int j = i; j < n; j++) {
      // duzina segmenta [i, j]
      int duzina = j - i + 1;

      // odredjujemo minimum i maksimum segmenta [i, j],
      // znajuci minimum i maksimum segmenta [i, j-1]
      if (a[j] > max)
        max = a[j];
      else if (a[j] < min)
        min = a[j];

      // provera da li se segment [i, j] moze urediti da sadrzi
      // uzastopne cele brojeve iz intervala [min, max]
      int raspon = max - min + 1;
      if (raspon == duzina)
```

```
      // ako je potrebno, azuriramo maksimalnu duzinu takvog
      // segmenta
      if (duzina > maxDuzina)
        maxDuzina = duzina;
    }
  }

  // prijavljujemo rezultat
  cout << maxDuzina << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Добри такмичари

Такмичари су се на једном турниру такмичили из програмирања и математике. Такмичар је добар ако не постоји такмичар који је од њега освојио строго више поена и из програмирања и из математике. Одредити број добрих такмичара.

Улаз: Са стандардног улаза се учитава број такмичара n ($2 \leq n \leq 50000$). У наредних n линија учитавају се по два броја (између 0 и 50000), раздвојена размаком, која представљају број поена такмичара из програмирања и математике.

Излаз: На стандардни излаз исписати број добрих такмичара.

Пример

Улаз	*Излаз*
9 2 4 9 3 7 6 3 7 7 2 3 9 4 9 6 4 6 8	5

Објашњење

Добри такмичари су $(9, 3)$, $(7, 6)$, $(3, 9)$, $(4, 9)$ и $(6, 8)$.

Решење

Груба сила

Решење грубом силом подразумева да се за сваког такмичара проанализирају сви други такмичари све док се не нађе неки који има строго више поена у обе дисциплине.

Сложеност најгорег случаја у овом приступу је јасно $O(n^2)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  // struktura u kojoj se mogu pamtiti poeni
  struct Poeni {
    int mat;
    int prog;
  };
```

```
  // ucitavamo poene svih takmicara
  int n;
  cin >> n;
  vector<Poeni> poeni(n);
  for (int i = 0; i < n; i++)
    cin >> poeni[i].mat >> poeni[i].prog;

  // ukupan broj do sada pronadjenih dobrih takmicara
  int brojDobrih = 0;
  // analiziramo jednog po jednog takmicara
  for (int i = 0; i < n; i++) {
    bool dobar = true;
    // uporedjujemo ga sa svim ostalim takmicarima
    for (int j = 0; j < n; j++) {
      // ne uporedjujemo ga samog sa sobom
      if (j == i) continue;
      // ako je neki drugi osvojio strogo vise poena u obe kategorije
      if (poeni[j].prog > poeni[i].prog && poeni[j].mat > poeni[i].mat) {
        // trenutni takmicar nije dobar
        dobar = false;
        break;
      }
    }
    // ako je trenutni takmicar dobar, uvecavamo broj dobrih
    if (dobar)
      brojDobrih++;
  }

  // ispisujemo ukupan broj dobrih takmicara
  cout << brojDobrih << endl;

  return 0;
}
```

Сортирање и инкрементално израчунавање низа максимума

Боље се решење може добити ако се низ на неки начин сортира. Претпоставимо да такмичаре сортирамо на основу поена из математике нерастући (за почетак претпоставимо да сви имају различит број поена). Тада за сваког такмичара лако можемо да одредимо оне који од њега имају више поена из математике (то су они у делу низа пре њега) као и оне који имају мање поена од њега из математике (то су они у делу низа после њега). Нико од такмичара у делу низа после њега не може да угрози његов статус доброг такмичара (јер нико нема строго више поена од њега из математике). Што се тиче такмичара у делу низа пре њега, ако претпоставимо да сви имају различит број поена из математике, неко од њих га угрожава само ако има строго више поена из програмирања од њега. Текући такмичар ће, дакле, бити добар ако и само ако нико испред њега у низу нема строго више поена из програмирања тј. ако и само ако је његов број поена из програмирања већи или једнак од броја поена свих такмичара испред њега у низу, што се своди на то да је његов број поена из програмирања већи или једнак од максималног броја поена из програмирања међу такмичарима који су у делу низа испред њега. Ако сваки пут тај максимум тражимо изнова, добијамо поново алгоритам квадратне сложености, што желимо да избегнемо.

Међутим, знамо да максимум можемо рачунати инкрементално. Наиме, можемо ојачати индуктивну хипотезу и претпоставити да приликом обраде текућег елемента знамо максимум броја поена из програмирања свих такмичара испред њега. Ту индуктивну хипотезу можемо лако одржати тиме што максимум ажурирамо на основу броја поена текућег такмичара.

Остаје још питање како обрадити случај када неколико такмичара има исти број поена из математике. Тај случај се једноставно решава ако такмичаре са истим бројем поена из математике обрађујемо у неопадајућем редоследу поена из програмирања. Тада ће текући максимални број поена из програмирања бити максимални број поена свих такмичара који имају већи или једнак број поена из математике и мањи или једнак број поена из програмирања од текућег.

- Текући такмичар може имати строго мање поена из програмирања од текућег максимума, само ако је тај максимум остварио неко ко има строго више поена из математике из њега (јер они који имају једнако поена као он из математике имају мање или једнако поена из програмирања) и тај такмичар не може бити добар.
- Са друге стране, ако текући такмичар има више или једнако поена из програмирања од текућег максимума, онда он јесте добар, јер нико ко има више строго више поена из програмирања од њега не може имати строго више поена и из математике (сви испред њега у низу имају мање или једнако поена из програмирања од текућег максимума, па и од њега, а сви иза њега имају мањи или једнак број поена из математике од њега).

Дакле, такмичаре можемо сортирати у нерастућем броју поена из математике, а оне који имају исти број поена из математике у неопадајућем броју поена из програмирања. Редом обилазимо сортирани низ и одржавамо максимум поена из програмирања до сада обрађених такмичара. Ако текући такмичар има више или једнако поена од текућег максимума, он је добар, па увећавамо број добрих такмичара и ажурирамо максимум.

На пример, поене такмичара можемо сортирати на следећи начин.

```
9 3
7 2
7 6
6 4
6 8
4 9
3 7
3 9
2 4
```

- Такмичар са скором $(9, 3)$ је добар, јер нико нема строго већи број поена из математике, а максимални број поена из програмирање након његове обраде је 3.
- Такмичар са скором $(7, 2)$ није добар (јер је 2 мање од максимума 3, па такмичар са скором $(9, 3)$ има већи број поена из оба предмета. Максимум остаје 3.
- Такмичар са скором $(7, 6)$ јесте добар (јер је 6 веће или једнако од максимума 3 , па међу претходним такмичарима који једини имају строгово више поена из математике, нико нема строго више поена из програмирања). Максимум постаје 6.
- Такмичар са скором $(6, 4)$ није добар (јер је 4 мање од максимума 6). Максимум остаје 6.
- Такмичар са скором $(6, 8)$ јесте добар (јер је 8 веће од досадашњег максимума 6). Максимум постаје 8.
- Такмичар са скором $(4, 9)$ јесте добар (јер је 9 веће од досадашњег максимума 8). Максимум постаје 9.
- Такмичар са скором $(3, 7)$ није добар (јер је 7 мање од максимума 9). Максимум остаје 9.
- Такмичар са скором $(3, 9)$ јесте добар (јер је 9 веће или једнако од максимума 9). Максимум остаје 9.
- Такмичар са скором $(2, 4)$ није добар (јер је 4 мање од максимума 9). Максимум остаје 9.

Сложеност доминантно потиче од сортирања и износи $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  // struktura u kojoj se mogu pamtiti poeni
  struct Poeni {
    int mat;
    int prog;
  };
```

```
  // ucitavamo poene svih takmicara
  int n;
  cin >> n;
  vector<Poeni> poeni(n);
  for (int i = 0; i < n; i++)
    cin >> poeni[i].mat >> poeni[i].prog;

  // sortiramo poene leksikografski,
  // nerastuce po matematici i neopadajuce po programiranju
  sort(begin(poeni), end(poeni),
       [](auto& p1, auto& p2) {
          return p1.mat > p2.mat ||
                 (p1.mat == p2.mat && p1.prog < p2.prog);
       });

  // ukupan broj do sada pronadjenih dobrih takmicara
  int brojDobrih = 0;
  // najveci do sada vidjen broj poena iz programiranja
  int maxProg = 0;
  for (int i = 0; i < n; i++) {
    // niko od do sada vidjenih nema strogo vise poena iz programiranja,
    // a niko od do sada nevidjenih nema strogo vise poena iz matematike
    if (poeni[i].prog >= maxProg) {
      // uvecavamo broj dobrih takmicara
      brojDobrih++;
      // azuriramo maksimalni broj osvojenih poena iz programiranja
      maxProg = poeni[i].prog;
    }
  }

  // ispisujemo ukupan broj dobrih takmicara
  cout << brojDobrih << endl;

  return 0;
}
```

2.11.3 Инкременталност - остале статистике

Задатак: Рутер

Дуж једне улице су равномерно распоређене зграде (растојање између сваке две суседне је једнако). За сваку зграду је познат број корисника које нови добављач интернета треба да повеже. Одредити у коју од зграда треба поставити рутер тако да би укупна дужина оптичких каблова којим се сваки од корисника повезује са рутером била минимална (рачунати само дужину каблова од зграде до зграде и занемарити дужине унутар зграда).

Улаз: У првом реду стандардног улаза налази се број n ($1 \leq n \leq 10^5$), а у наредном n природних бројева раздвојених размацима који представљају број корисника у свакој од n зграда.

Излаз: На стандардни излаз исписати минималну дужину каблова.

Пример

Улаз	*Излаз*
6 3 5 1 6 2 4	30

Решење

Груба сила

Наивно решење би подразумевало да се израчуна дужина каблова за сваку могућу позицију рутера и да се одабере најмањи. Да бисмо израчунали дужину каблова, ако је рутер у згради на позицији k, рачунамо заправо

збир

$$\sum_{i=0}^{n-1} |k-i| \cdot a_i,$$

где је a_i број корисника у згради i.

Сваки тежински збир можемо израчунати у времену $O(n)$, па пошто се испитује n позиција, алгоритам је сложености $O(n^2)$.

```
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> stanara(n);
  for (int i = 0; i < n; i++)
    cin >> stanara[i];

  long long min_duzina_kablova = numeric_limits<long long>::max();
  // obrađujemo sve zgrade od 1 do n-1
  for (int k = 0; k < n; k++) {
    long long duzina_kablova = 0;
    for (int i = 0; i < k; i++)
      duzina_kablova += (k - i) * stanara[i];
    for (int i = k+1; i < n; i++)
      duzina_kablova += (i - k) * stanara[i];

    if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
  }

  cout << min_duzina_kablova << endl;

  return 0;
}
```

Решење на основу принципа инкременталности

Много боље решење и линеарни алгоритам можемо добити ако применимо принцип инкременталности и избегнемо рачунање у сваком кораку из почетка. Размотримо како се дужина каблова мења када се рутер помера са зграде k на зграду $k+1$.

Дужину каблова за рутер у згради $k+1$ добијамо од дужине каблова за рутер у згради k тако што ту дужину увећамо за укупан број станара закључно са зградом k и умањимо је за укупан број станара почевши од зграде $k+1$. То је заправо интуитивно прилично јасно и без компликованог математичког извођења. Померањем рутера за дужину једне зграде надесно, сваком станару који живи закључно до зграде k дужина кабла се повећала за једно растојање између зграда, а свим станарима од зграде $k+1$ надесно се та дужина смањује за једно растојање између зграда.

Формално, математички, то се може показати на следећи начин. Ако је рутер на позицији k, тада је дужина каблова једнака

$$d_k = \sum_{i=0}^{k-1} (k-i) \cdot a_i + \sum_{i=k+1}^{n-1} (i-k) \cdot a_i.$$

Ако је рутер на позицији $k+1$, тада је дужина каблова једнака

$$d_{k+1} = \sum_{i=0}^{k}(k+1-i)\cdot a_i + \sum_{i=k+2}^{n-1}(i-k-1)\cdot a_i.$$

Разлика између те две суме једнака је

$$\begin{aligned}
d_{k+1} - d_k &= \left(\sum_{i=0}^{k}(k+1-i)\cdot a_i - \sum_{i=0}^{k-1}(k-i)\cdot a_i\right) + \\
&\quad \left(\sum_{i=k+2}^{n-1}(i-k-1)\cdot a_i - \sum_{i=k+1}^{n-1}(i-k)\cdot a_i\right) \\
&= \left(\sum_{i=0}^{k-1}((k+1-i)-(k-i))\cdot a_i\right) + a_k - a_{k+1} + \\
&\quad \left(\sum_{i=k+2}^{n-1}((i-k-1)-(i-k))\cdot a_i\right) \\
&= \sum_{i=0}^{k-1} a_i + a_k - a_{k-1} - \sum_{i=k+2}^{n-1} a_i \\
&= \sum_{i=0}^{k} a_i - \sum_{i=k+1}^{n-1} a_i
\end{aligned}$$

Укупне бројеве станара пре и после дате зграде можемо такође рачунати инкрементално (при преласку на наредну зграду, први број се увећава, а други умањује за број станара текуће зграде). Слично смо радили и у задатку Оптимални сервис.

Дакле, у програму можемо да памтимо три ствари: дужину каблова d_k ако је рутер на позицији k, укупан број станара pre_k пре зграде k (не укључујући њу) и укупан број станара $posle_k$ од зграде k (укључујући њу) до краја. На почетку, када је $k = 0$, први број d_0 морамо експлицитно израчунати као $\sum_{i=1}^{n-1} i \cdot a_i$, други број треба иницијализовати на нулу $pre_0 = 0$, а трећи на укупан број свих станара $posle_k = \sum_{i=0}^{n-1} a_i$. Затим за свако k од 1 до $n-1$ рачунамо $pre_k = pre_{k-1} + a_{k-1}$, $posle_k = posle_{k-1} - a_{k-1}$ и затим $d_k = d_{k-1} + pre_k - posle_k$.

Илуструјмо извршавање овог алгоритма на примеру зграда у којима живи редом $3, 5, 1, 6, 2, 4$ станара.

```
k    dk  pre_k   posle_k
0    53   0       21
1    38   3       18
2    33   8       13
3    30   9       12
4    39  15        6
5    52  17        4
```

Приметимо да је могуће извршити и малу оптимизацију (додуше која неће поправити асимптотску сложеност) на основу монотоности низа d_k и петљу прекинути чим се број d_k први пут повећа. Наиме, вредности у том низу ће опадати до тражене минималне вредности, након чега ће кренути да расту. У претходном примеру, могли смо закључити да је минимална дужина каблова 30, чим се у наредном кораку та вредност повећала на 39.

Пошто је и за једну и за другу фазу потребно време $O(n)$, то је уједно сложеност овог алгоритма.

Интересантно, укупну почетну дужину каблова можемо израчунати ефикасно и без множења, тако што на збир додамо број станара у последњој згради, затим број станара у последње две зграде, затим број станара у последње три зграде и тако даље, све док не додамо број станара у свим зградама осим прве.

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> stanara(n);
  for (int i = 0; i < n; i++)
    cin >> stanara[i];

  // krećemo od zgrade 0
  // ukupna dužina kablova ako je ruter u tekućoj zgradi
  long long duzina_kablova = 0;
  for (int i = 0; i < n; i++)
    duzina_kablova += stanara[i] * i;
  // broj stanara pre tekuće zgrade
  long long stanara_pre = 0;
  // broj stanara od tekuće zgrade do kraja
  long long stanara_posle = 0;
  for (int i = 0; i < n; i++)
    stanara_posle += stanara[i];

  // minimalna duzina kablova
  long long min_duzina_kablova = duzina_kablova;

  // obrađujemo sve zgrade od 1 do n-1
  for (int k = 1; k < n; k++) {
    // ažuriramo brojeve stanara
    stanara_pre += stanara[k-1];
    stanara_posle -= stanara[k-1];
    // ažuriramo duz
    duzina_kablova += stanara_pre - stanara_posle;
    if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
  }

  cout << min_duzina_kablova << endl;

  return 0;
}
```

Задатак: Централа

Дата је матрица са подацима о томе колико каблова је потребно увести у блокове зграда у неком граду (сваки блок је представљен једним елементом матрице). Каблови се могу постављати само паралелно са улицама, па се растојање између два блока рачуна као $|x_1 - x_2| + |y_1 - y_2|$, где су (x_1, y_1) и (x_2, y_2) координате блокова. Наћи блок који је најпогоднији за централу, тј. онај блок коме је збир растојања до осталих блокова, помножених бројем потребних каблова, најмањи.

Улаз: У првом реду стандардног улаза налазе се бројеви V и K раздвојени једним размаком, број врста и колона матрице ($1 \leq V \leq 10$, $1 \leq K \leq 10$). У наредних V редова налази се по K природних бројева, који нису већи од 1000. Сваки од тих бројева представља број каблова које треба довести до одговарајућег блока зграда.

Излаз: У првом реду исписати редни број реда, а у другом редни број колоне (бројећи од 0) оног блока, за који је поменути тежински збир растојања до осталих блокова најмањи.

У случају да одговор није једнозначан, предност има блок са мањом првом координатом, а међу блоковима

са истом првом координатом предност има блок са мањом другом координатом.

Пример

Улаз	*Излаз*
3 4 2 1 1 4 1 3 2 1 2 2 2 1	1 1

Решење

Зе ефикасно решавање овог задатка је кључно да се увиди да се проблем своди на два једнодимензиона задатка, слична задатку Рутер.

Заиста, на одређивање врсте у коју треба сместити централу утичу само разлике y координата (тј. редних бројева врста) појединих блокова. Исто тако, тражена колона зависи само од разлика x координата (тј. редних бројева колона) појединих блокова.

Према томе, довољно је користити два низа, који ће садржати збирове врста, односно збирове колона дате матрице (цела матрица није потребна).

За одређивање тражене врсте довољно је анализирати низ збирова врста. Слично томе, за одређивање тражене колоне довољно је анализирати низ збирова колона, и то на потпуно исти начин. Зато је згодно поступак проналажења најповољније координате сместити у функцију и позвати два пута, једном за врсте а други пут за колоне.

За објашњење самог поступка одређивања оптималне координате види задатак Рутер. Поступак се овде разликује само у томе што се уместо минималног броја потребних каблова тражи први (најмањи) индекс на ком се тај минимум достиже.

```
#include <iostream>
#include <vector>

using namespace std;

int Sredina(vector<int> a) {
    long long duzinaKablova = 0;
    int n = (int)a.size();
    for (int i = 0; i < n; i++)
        duzinaKablova += a[i] * i;
    long long brZgradaPre = 0;
    long long brZgradaPosle = 0;
    for (int i = 0; i < n; i++)
        brZgradaPosle += a[i];

    long long minDuzinaKablova = duzinaKablova;
    int najboljeK = 0;

    for (int k = 1; k < n; k++) {
        brZgradaPre += a[k - 1];
        brZgradaPosle -= a[k - 1];
        duzinaKablova += brZgradaPre - brZgradaPosle;
        if (duzinaKablova < minDuzinaKablova) {
            minDuzinaKablova = duzinaKablova;
            najboljeK = k;
        }
    }
    return najboljeK;
}

int main() {
  int v, k;
  cin >> v >> k;
```

```
  vector<int> zbirVrste(v);
  vector<int> zbirKolone(k);
  for (int i = 0; i < v; i++) {
    for (int j = 0; j < k; j++) {
      int x;
      cin >> x;
      zbirVrste[i] += x;
      zbirKolone[j] += x;
    }
  }

  cout << Sredina(zbirVrste) << endl;
  cout << Sredina(zbirKolone) << endl;

  return 0;
}
```

Задатак: Највећи тежински збир после цикличног померања

Дат је низ a целих бројева дужине n. Дозвољена је операција цикличног померања тј. ротације низа улево за једно место, операцију можемо понављати произвољан број пута. Написати програм којим се одређује најмањи број померања улево тако да тежински збир

$$0 \cdot a_0 + 1 \cdot a_1 + 2 \cdot a_2 + 3 \cdot a_3 + \ldots + (n-1) \cdot a_{n-1},$$

по модулу 1234567 у трансформисаном низу има највећу могућу вредност.

Улаз: У првој линији стандардног улаза налази се природан број n ($n \leq 50000$). У следећих n линија налазе се редом елементи низа a (цели бројеви из интервала $[0, 100]$).

Излаз: На стандардном излазу у једној линији приказати тражени број цикличних померања улево (цео број од 0 до $n-1$), а у наредној највећу вредност тежинског збира.

Пример

Улаз	*Излаз*
3	2
5	13
4	
1	

Решење

Груба сила

Задатак можемо решити тако што $n-1$ пут низ ефективно ротирамо за по једно место улево (како смо приказали у задатку Циклично померање за једно место) израчунавајући сваки пут тежински збир изнова, тражећи максимум и позицију максимума тако добијених тежинских збирова (уобичајеним алгоритмом, приказаним, на пример, у задатку Редни број максимума).

Пошто се тежински збирови у нашем задатку рачунају по датом модулу mod, све операције у претходним формулама треба заменити одговарајућим операцијама по модулу (њих можемо реализовати у засебним функцијама, на начин који смо приказали у задатку Операције по модулу и Монопол).

Максимална вредност тежинског збира се постиже када низ има 50000 елемената чија је вредност 100. Тада се са 100 множи сваки елемент од 0 до 49999 и максимални тежински збир је једнак 100 пута вредност збира свих природних бројева до 49999 што је једанко $100 \cdot \frac{49999 \cdot (49999+1)}{2}$, што је око $1.25 \cdot 10^{11}$ и стаје у опсег 64-битног типа (у језику C++ то је тип `long long`). Ако тај тип употребимо за чување збира, онда остатак при дељењу можемо израчунати само једном, након целокупног израчунавања збира, чиме се добија на ефикасности.

Пошто број операција потребан за израчунавање тежинског збира линеарно зависи од дужине низа n (сваки тежински збир се израчунава у сложености $O(n)$), овај приступ доводи до квадратне сложености алгоритма

(сложености $O(n^2)$). Чак иако бисмо оптимизовали број израчунавања остатака, програм ће бити неефикасан услед квадратне сложености алгоритма и пуно померања елемената низа.

```
#include <iostream>
#include <vector>

using namespace std;

// zbir brojeva x i y po modulu mod
int zm(int x, int y, int mod) {
  return (x % mod + y % mod) % mod;
}

// ciklicno pomeranje (rotiranje) niza za jedno mesto ulevo
void rotirajUlevo(vector<int>& a, int n) {
  int pom = a[0];
  for (int i = 0; i < n - 1; i++)
    a[i] = a[i + 1];
  a[n - 1] = pom;
}

// suma brojeva a[i]*i, za i od 0 do n-1, po modulu mod
int tezinskaSuma(const vector<int>& a, int n, int mod) {
  int s = 0;
  for (int i = 0; i < n; i++)
    s = zm(s, i * a[i], mod);
  return s;
}

int main() {
  // ubrzavamo ucitavanje elemenata niza
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // modul dat u tekstu zadatka
  const int mod = 1234567;

  // maksimum inicijalizujemo na tezinsku sumu pocetnog niza
  int maxTezinskaSuma = tezinskaSuma(a, n, mod);
  // redni broj maksimuma inicijalizujemo na 0
  int maxBrojRotacija = 0;
  for (int i = 1; i < n; i++) {
    // rotiramo elemente niza a za jedno mesto ulevo
    rotirajUlevo(a, n);
    // izracunavamo novu tezinsku sumu
    int tekSuma = tezinskaSuma(a, n, mod);
    // azuriramo podatke o maksumumu, ako je to potrebno
    if (tekSuma > maxTezinskaSuma) {
      maxTezinskaSuma = tekSuma;
      maxBrojRotacija = i;
    }
  }
  // prijavljujemo redni broj i vrednost maksimuma
```

```
  cout << maxBrojRotacija << endl;
  cout << maxTezinskaSuma << endl;
  return 0;
}
```

Избегавање ротација низа

Уместо ефективне ротације свих елемената низа, ефекат обиласка низа који је ротиран за k места улево можемо постићи тако што обилазак крећемо од позиције k, а затим у петљи која има n итерација увећавамо бројач за 1, али овај пут по модулу n (када бројач постане n вредност му се враћа на нулу што можемо постићи било експлицитним испитивањем вредности након сваког увећања бројача, било израчунавањем остатка при дељењу са n, што је спорије од гранања, јер је израчунавање остатка при дељењу обично релативно скупа операција).

Иако избегавамо ротације, свака од n тежинских сума се израчунава засебно у времену $O(n)$, па сложеност остаје $O(n^2)$.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ubrzavamo ucitavanje elemenata niza
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // modul dat u tekstu zadatka
  const int mod = 1234567;

  // maksimum inicijalizujemo na tezinsku sumu pocetnog niza
  int maxBrojRotacija = 0;
  long long maxTezinskaSuma = 0;

  // redni broj maksimuma inicijalizujemo na 0
  for (int i = 0; i < n; i++) {
    // koristimo tip long long, da ne bismo morali da izracunavamo
    // ostatak posle svakog sabiranja, vec samo nakon izracunavanja
    // celog zbira
    long long tekSuma = 0;
    int k = i;
    for (int j = 0; j < n; j++) {
      tekSuma += a[k] * j;
      if (++k == n) k = 0;
    }
    tekSuma %= mod;
    // azuriramo podatke o maksumumu, ako je to potrebno
    if (tekSuma > maxTezinskaSuma) {
      maxTezinskaSuma = tekSuma;
      maxBrojRotacija = i;
    }
  }
  // prijavljujemo redni broj i vrednost maksimuma
  cout << maxBrojRotacija << endl;
```

```
  cout << maxTezinskaSuma << endl;
  return 0;
}
```

Још једна техника којом можемо избећи ротацију је да алоцирамо двоструко више меморије, да елементе оригиналног низа сместимо два пута, једном иза другог и да затим ефекат ротације за k места постижемо тако што обилазимо n елемената тако проширеног низа почевши од позиције k.

Овим добијамо на брзини, али губимо на заузећу меморије (додуше, не асимптотски, јер временска сложеност остаје $O(n^2)$, а меморијска $O(n)$).

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ubrzavamo ucitavanje elemenata niza
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(2*n);
  for (int i = 0; i < n; i++){
    cin >> a[i];
    // prepisujemo niz iza originala, cime simuliramo clanove niza
    // rasporedjene u krug
    a[i+n] = a[i];
  }

  // modul dat u tekstu zadatka
  const int mod = 1234567;

  // maksimum inicijalizujemo na tezinsku sumu pocetnog niza
  int maxBrojRotacija = 0;
  long long maxTezinskaSuma = 0;

  // redni broj maksimuma inicijalizujemo na 0
  for (int i = 0; i < n; i++) {
    // koristimo tip long long, da ne bismo morali da izracunavamo
    // ostatak posle svakog sabiranja, vec samo nakon izracunavanja
    // celog zbira
    long long tekSuma = 0;
    for (int j = 0; j < n; j++)
      tekSuma += a[i + j] * j;
    tekSuma %= mod;
    // azuriramo podatke o maksumumu, ako je to potrebno
    if (tekSuma > maxTezinskaSuma) {
      maxTezinskaSuma = tekSuma;
      maxBrojRotacija = i;
    }
  }
  // prijavljujemo redni broj i vrednost maksimuma
  cout << maxBrojRotacija << endl;
  cout << maxTezinskaSuma << endl;
  return 0;
}
```

Оптимизација на основу инкременталности

Задатак је могуће решити и много ефикасније ако применимо принцип инкременталности тј. пронађемо начин да тежински збир после ротације ефикасно израчунамо на основу познатог тежинског збира пре ротације. Можемо уочити да је у тежинском збиру после померања улево сваки елемент низа, изузев првог елемента a_0, један пут мање укључен него пре померања, а први елемент је укључен $n-1$ пут. Обележимо са z_i тежински збир добијен приликом померања полазног низа улево i пута, а са z класичан збир свих елемената низа. Према томе важе следеће једнакости:

$$\begin{aligned} z_0 &= 0 \cdot a_0 + 1 \cdot a_1 + 2 \cdot a_2 + ... + (n-2) \cdot a_{n-2} + (n-1) \cdot a_{n-1} \\ z_1 &= 0 \cdot a_1 + 1 \cdot a_2 + 2 \cdot a_3 + ... + (n-2) \cdot a_{n-1} + (n-1) \cdot a_0 \\ z_2 &= 0 \cdot a_2 + 1 \cdot a_3 + 2 \cdot a_4 + ... + (n-2) \cdot a_0 + (n-1) \cdot a_1 \\ &\dots \\ z_{n-1} &= 0 \cdot a_{n-1} + 1 \cdot a_0 + 2 \cdot a_1 + ... + (n-2) \cdot a_{n-3} + (n-1) \cdot a_{n-2} \end{aligned}$$

$$z = a_0 + a_1 + ... + a_{n-2} + a_{n-1}$$

Приметимо да важи

$$\begin{aligned} z_0 - z_1 &= a_1 + a_2 + \ldots + a_{n-1} - (n-1) \cdot a_0 = z - n \cdot a_0 \\ z_1 - z_2 &= a_2 + a_3 + \ldots + a_0 - (n-1) \cdot a_1 = z - n \cdot a_1 \\ &\dots \\ z_{n-2} - z_{n-1} &= a_{n-1} + a_0 + \ldots + a_{n-3} - (n-1) \cdot a_{n-2} = z - n \cdot a_{n-2} \end{aligned}$$

итд.

Према томе $z_{i-1} - z_i = z - n \cdot a_{i-1}$, тј.

$$z_i = z_{i-1} - z + n \cdot a_{i-1}.$$

Дакле, тежински збир после померања за једно место улево можемо једноставно израчунати без померања низа на основу тежинског збира низа пре померања и збира свих елемената низа.

Имплементацију можемо извршити на следећи начин. Израчунамо тежински збир полазног низа $z_0 = \sum_{i=0}^{n-1} i \cdot a_i$ и класични збир свих елемената низа $z = \sum_{i=0}^{n-1} a_i$ (једноставним алгоритмом сабирања елемената низа). Од свих збирова треба израчунати највећи и одредити после колико ротација се та највећи збир постиже, што, наравно, радимо уобичајеним алгоритмом одређивања позиције максимума серије елемената (приказаним, на пример, у задатку Редни број максимума). Максимум ћемо иницијализовати на први израчунати тежински збир, а број померања ћемо иницијализовати на 0. Затим рачунамо тежинске збирове за низове добијене померањем низа за једно место улево i пута, и то редом за i од 1 до $n-1$. Тежински збир z_i за низ добијен након i померања рачунамо тако што претходни тежински збир z_{i-1} умањимо за збир свих елемената z и увећамо за $n \cdot a_{i-1}$. Проверавамо да ли је добијени тражени збир већи од дотадашњег максимума и ако јесте коригујемо максимум и број померања.

Приметимо да време потребно за израчунавање почетних збирова линеарно зависи од дужине низа n, док је за израчунавање сваког наредног збира довољан константан број операција, тако да је укупна временска сложеност алгоритма линеарна тј. $O(n)$.

```
#include <iostream>
#include <vector>

using namespace std;

// zbir x i y po modulu mod
int zm(int x, int y, int mod) {
  return (x % mod + y % mod) % mod;
}

// razlika x i y po modulu mod
int rm(int x, int y, int mod) {
  return (x % mod - y % mod + mod) % mod;
```

```
}

int main() {
  // ubrzavamo ucitavanje elemenata niza
  ios_base::sync_with_stdio(false);

  // ucitavamo broj elemenata niza
  int n;
  cin >> n;
  // ucitavamo niz
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  const int mod = 1234567;

  // izracunavamo tezinsku sumu i*a[i] i obicnu sumu elemenata a[i] po
  // modulu mod
  int tezinskaSuma = 0;
  int suma = 0;
  for (int i = 0; i < n; i++) {
    tezinskaSuma = zm(tezinskaSuma, i * a[i], mod);
    suma = zm(suma, a[i], mod);
  }

  // najveca do sada vidjena tezinska suma
  int maxTezinskaSuma = tezinskaSuma;
  // broj rotacija kojima se postize najveca tezinska suma
  int maxBrojPomeranja = 0;
  for (int i = 1; i < n; i++) {
    // rotacija za jedno mesto ulevo na sledeci nacin menja tezinsku sumu
    tezinskaSuma = zm(rm(tezinskaSuma, suma, mod), n * a[i - 1], mod);

    // proveravamo da li je dobijena tezinska suma veca od do tada
    // najvece
    if (tezinskaSuma > maxTezinskaSuma) {
      maxTezinskaSuma = tezinskaSuma;
      maxBrojPomeranja = i;
    }
  }

  // ispisujemo rezultat
  cout << maxBrojPomeranja << endl;
  cout << maxTezinskaSuma << endl;
  return 0;
}
```

Задатак: Дрва

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање и инкременталност

Једна идеја за решење задатка је да се пронађе највиша висина тестере која је једнака висини неког дрвета са којом се добија довољно дрвета. Нека је то висина H и нека се њом добија количина $D \geq D_p$, где је D_p потребна количина и нека се на тој висини сече k стабала. Нама је потребно да подигнемо тестеру још мало тако да се вишак од $D - D_p$ елиминише. Сигурни смо да ће подизање да буде мање од висине наредног дрвета

по висини, јер када се тестера дигне на висину наредног дрвета, нећемо имати довољно насеченог дрвета (јер је H највиша висина дрвета за коју је насечено довољно). Ово значи да се тим финим подешавањем висине неће променити број дрва која се секу. Дакле, ако је $H+h$ мање од висине следећег дрвета, тада ће повећање висине са H на $H+h$ метара смањити исечену количину дрвета за $k \cdot h$ метара (јер тестера и даље сече истих k стабала). Dakle, potrebno je pronaći maksimalnu vrednost h takvu da je $D - h \cdot k \geq D_p$, tj. da je $h \leq \frac{D-D_p}{k}$, а то је вредност $h = \left\lfloor \frac{D-D_p}{k} \right\rfloor$. Висина је тада $H + h$.

Потребно је само бити обазрив и посебно рамзотрити случај када се ни постављањем тестере на висину најнижег дрвета не добије довољно дрвета (а то је могуће да се деси). Тада је тестеру потребно спустити до земље, а онда поправку израчунати на горе описани начин. Најлакши начин да се ово имплементира је да се у низ вештачки дода посебно дрво висине 0.

Остаје још питање како одредити количину дрвета која се добија када се тестера постави на висину дрвета на позицији k. Један начин је да се та количина сваки пут изнова рачуна, но то би било веома неефикасно. Боље решење је да се низ сортира и да се затим количина рачуна инкрементално. Спуштањем секире на висину дрвета на позицији k од сваког од првих k дрвета одсечено је парче од $h_k - h_{k-1}$ метара, па ако знамо количину дрвета која се исече када је тестера на висини дрвета на позицији $k-1$, количину дрвета када је тестера на висини дрвета на позицији k можемо веома једноставно добити увећавањем те количине за $k \cdot (h_k - h_{k-1})$.

Прикажимо рад овог алгоритма на примеру из поставке задатка. Потребно је да насечемо 14 метара дрвета, а висине су $[24, 21, 19, 14, 22]$. Након сортирања овог низа опадајуће, добијамо низ висина $[24, 22, 21, 19, 14, 0]$.

- Ако је висина тестере 24, насећи ћемо 0 метара.
- Ако је висина тестере 22, количина ће се са 0 повећати за $1 \cdot (24 - 22)$ тј. за 2 и биће једнака 2.
- Ако је висина тестере 21, количина ће се повећати за $2 \cdot (22 - 21)$ тј. за 2 и биће једнака 4.
- Ако је висина тестере 19, количина ће се повећати за $3 \cdot (21 - 19)$ тј. за 6 и биће једнака 10.
- Ако је висина тестере 14, количина ће се повећати за $4 \cdot (19 - 14)$ тј. за 20 и биће једнака 30. Ово је више него што нам је потребно, па ћемо додатно мало подигнути тестеру. Вишак у односу на потребну количину је $30 - 14 = 16$. Ако то поделимо на 4 дрвета која се секу добијамо $\left\lfloor \frac{20-14}{4} \right\rfloor = 4$ и висину подешавамо на $14 + 4 = 18$.

Сортирање дрвећа по висини је сложености $O(n \log n)$. Након тога се дрво чија је висина довољна да би се добила довољна количина насеченог дрвета одређује једним проласком кроз низ и инкреметанталним ажурирањем количине насеченог дрвета (што је сложености $O(1)$), па се тражено дрво налази у сложености $O(n)$. Завршна поправка висине врши се у времену $O(1)$. Дакле, сложеношћу доминира сортирање и укупна сложеност износи $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int testera(vector<int>& visina, long long potrebno) {
  // sortiramo drva po visini, opadajuće
  sort(visina.begin(), visina.end(), greater<int>());

  // nasečena količina drva
  long long naseceno = 0;
  // broj drva koja sečemo
  int k;
  // spuštamo testeru do visine narednog narednog drveta
  for (k = 1; k < visina.size(); k++) {
    // računamo koliko je drveta nasečeno ako je testera na visini
    // drveta na poziciji k - testera tada zaseca tačno k stabala
    naseceno += (visina[k-1] - visina[k]) * k;
    // prvi put kada nasečemo potrebnu količinu prekidamo petlju
    if (naseceno >= potrebno)
      break;
  }
```

```
  // popravljamo malo visinu (fino štelujući visinu između visine
  // drveta na poziciji k-1 i drveta na poziciji k) tako da uzmemo što
  // manje, ali opet dovoljno drveta
  return visina[k] + (naseceno - potrebno) / k;
}

int main() {
  int n;
  cin >> n;
  vector<int> visina(n + 1);
  visina[0] = 0;
  for (int i = 1; i <= n; i++)
    cin >> visina[i];
  long long potrebno;
  cin >> potrebno;

  cout << testera(visina, potrebno) << endl;

  return 0;
}
```

Задатак: Хиршов h-индекс

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Инкрементално бројање чланака

Ефикасан алгоритам можемо добити ако вредности B_h за разне h израчунавамо на основу принципа инкременталности. Можемо приметити да се број радова B_h који имају бар h цитата може израчунати као збир броја радова који имају више од h цитата тј. броја радова који имају бар $h+1$ цитат и броја радова који имају тачно h цитата, тј. $B_h = B_{h+1} + b_h$, где је b_h број радова који имају тачно h цитата. Сличну технику инкременталног израчунавања збира већ смо употребљавали (на пример, у задатку Префикс највећег збира).

Дакле, одговара нам да претрагу организујемо унатраг и да за сваки број h од n па уназад до нуле одређујемо број радова који имају бар h цитата, заустављајући се када пронађемо прву вредност h тако да је $B_h \geq h$. Остаје питање како израчунати вредности b_h и како израчунати почетну вредност B_n. То можемо урадити у једном пролазу кроз низ c (заправо, низ c не морамо ни памтити, већ приликом учитавања његових чланова можемо само израчунавати елементе b_h за $0 \leq h < n$, и уједно израчунати вредност B_n). Ове вредности можемо памтити у низу. Низ има $n+1$ елеменат, при чему се на позицијама од 0 до $n-1$ израчунавају вредности b_h, а на позицији n се чува вредност B_n. Низ иницијализујемо на нулу и сваки пут када учитамо број цитата неког рада проверавамо да ли је мањи од n и ако јесте, увећавамо бројач чланака са тим бројем цитата (број b_h), а у супротном увећавамо број чланака са бар n цитата (број B_n).

На пример, ако је низ цитата `3 5 12 7 5 9 0 17`, тада се низ попуњава на следећи начин.

```
b0  b1  b2  b3  b4  b5  b6  b7  B8
1   0   0   1   0   2   0   1   3
```

Имамо 3 рада са 8 и више цитата, 1 рад са тачно 7 цитата, 2 рада са тачно 5 цитата, један рад са тачно 3 цитата и један рад без цитата.

На основу претходног низа инкременталним сабирањем уназад лако израчунавамо све вредности B_h (додуше, нису нам све потребне).

```
B0  B1  B2  B3  B4  B5  B6  B7  B8
8   7   7   7   6   6   4   4   3
```

Израчунавање заправо тече овако.

- Вредност B_8 једнака је 3, што значи да постоји тачно 3 рада са 8 и више цитата, па је h-индекс мањи од 8.
- Вредност B_7 једнака је $B_8 + b_7 = 3 + 1 = 4$, што значи да постоји тачно 4 рада са 7 или више цитата, па је h-индекс мањи од 7.
- Вредност $B_6 = B_7 + b_6 = 4 + 0 = 4$ нам говори да постоји тачно 4 рада са 6 или више цитата, па је h-индекс мањи и од 6.
- Вредност $B_5 = B_6 + b_5 = 4 + 2 = 6$ нам говори да постоји тачно 6 радова са 5 или више цитата, што нам говори да h-индекс јесте једнак 5.

Сложеност овог приступа је линеарна $O(n)$ (уз ангажовање низа од $n + 1$ елемент) и донекле је боља него у случају када се користи сортирање. Ако је број цитата сваког рада дат низом који се не сме мењати, неопходно је користити помоћни низ димензије $n + 1$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  // broj clanaka
  int n;
  cin >> n;

  // broj_clanaka[i] = broj clanaka koji imaju tacno i citata
  // broj_clanaka[n] = broj clanaka koji imaju >= n citata
  vector<int> broj_clanaka(n + 1, 0);
  for (int i = 0; i < n; i++) {
    // ucitavamo broj citata tekuceg clanka
    int broj_citata;
    cin >> broj_citata;
    // azuriramo statistiku o broju clanaka
    broj_clanaka[broj_citata < n ? broj_citata : n]++;
  }

  // pokusavamo da uspostavimo sto veci h_indeks
  int h_indeks = n;
  // ukupno_clanaka = broj clanaka koji imaju >= h_indeks citata
  int ukupno_clanaka = broj_clanaka[n];
  while (ukupno_clanaka < h_indeks) {
    // moramo da smanjimo h_indeks
    h_indeks--;
    // azuriramo broj clanaka sa >= h_indeks citata
    ukupno_clanaka += broj_clanaka[h_indeks];
  }

  // vazi da je broj clanaka sa >= h_indeks citata >= h_indeks i
  // h_indeks je najveci broj za koji to vazi, pa je on trazeni
  // rezultat
  cout << h_indeks << endl;

  return 0;
}
```

Задатак: Максимални збир сегмента фиксираног почетка

Дат је низ целих бројева дужине n. За фиксирани леви крај $0 \leq i < n$, потребно је пронаћи најмањи збир облика $a_i + a_{i+1} + \ldots + a_j$, где је $i \leq j < n$. Напиши програм који учитава низ, а затим и различите вредности почетка i и за сваку вредност почетка исписује тражени максималан збир.

Улаз: Са стандардног улаза се учитава број n ($3 \leq n \leq 10^5$), а затим у наредном реду n елемената низа раздвојених размацима. Након тога се учитава број упита q ($1 \leq q \leq 10^5$), а затим и q природних бројева i ($0 \leq i < n$), раздвојених размацима.

Излаз: За сваку унету вредност i, у посебној линији стандардног излаза исписати тражени максималан збир.

Пример

Улаз	*Излаз*

`5`
`3 -1 4 -3 2`
`3`
`0 2 3`
	`6 4 -1`

2.11.4 Покретни прозор

Задатак: Пермутоване подниске

Дате су две ниске s и x, састављене искључиво од декадних цифара. Написати програм којим се одређује број подниски (узастопних карактера) ниске s од чијих се цифара може формирати ниска x тако да сваку узету цифру употребимо тачно једанпут.

Улаз: Прва линија стандардног улаза садржи ниску s, а друга линија ниску x.

Излаз: На стандардном излазу приказати у једној линији тражени број начина.

Пример

Улаз	*Излаз*
`5242245422` `242`	`4`

Објашњење: ниску `242` можемо формирати узимајући по 3 узастопне цифре из ниске `5242245422` редом од позиција 1, 2, 3 и 7.

Решење

Груба сила - поређење свих подниски

Главни алгоритам у великој мери одговара оном који смо употребили у задатку Подниске.

Редом ћемо поредити све подниске ниске `s` дужине једнаке дужини ниске `x` са ниском `x`. Набрајање подниски можемо остварити петљом која набраја све почетне позиције подниске од нуле, па све док се подниска налази унутар ниске (тј. док је њена последња позиција која је једнака збиру почетка и дужине ниске `x` умањеном за један мања од дужине ниске `s`).

Међутим, поређење ниски није класично и једно од основних питања у овом задатку је како проверити да ли се две ниске поклапају до на пермутацију карактера, тј. да ли се евентуалном променом редоследа карактера једне ниске може добити друга. У задатку Провера пермутација видели смо да постоји неколико начина да се то уради. Један је да се провери да ли оба стринга након сортирања дају исти резултат, а други је био да се провери да ли је мултискуп карактера једнак, при чему за репрезентацију мултискупа можемо употребити било низ бројача који сваком карактеру додељује број његових појављивања у ниски, било неку библиотечку колекцију којом се може репрезентовати мултискуп. У свим решењима описаним у наставку овог задатка разматраћемо опцију пребројавања карактера.

Једна могућност је да експлицитно издвајамо једну по једну подниску. У језику C++ подниску можемо издвојити методом `substr` којој су параметри почетак и дужина подниске.

Тада можемо дефинисати функцију која проверава да ли су две дате ниске једнаке до на пермутацију карактера. У њој се броје карактери једне и друге дате ниске и затим се добијени низови бројача упоређују.

Пошто у овом задатку знамо да се обе ниске састоје само од цифара, онда низ бројача можемо реализовати као десетоелементни низ (на позицији нула чуваћемо број појављивања карактера нула, на позицији један чуваћемо број појављивања карактера један итд., што лако можемо остварити одузимањем кода карактера нула од кода текућег карактера, слично као што је показано у задатку Фреквенција знака). Рецимо да би се за ово могла употребити и мапа тј. речник, поготово ако скуп карактера који се могу јавити у нискама не би био унапред познат.

Ако уместо класичног низа користимо `vector`, тада проверу једнакости можемо веома једноставно извршити оператором `==`.

Обележимо са S дужину ниске `s`, а са X дужину ниске `x`. Проверава се око $S - X$ подниски, што је, под претпоставком да је s доста дуже него x, једнако $O(S)$. Пребројавање карактера сваке подниске врши се у времену $O(X)$, исто као и поређење једнакости две подниске (при чему смо због малог броја бројача, претпоставили да су и ажурирање и поређење низова бројача операције константне сложености). Под датим претпоставкама, сложеност се може проценити са $O(S \cdot X)$.

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

// prebrojava koliko se puta u stringu s javlja svaka od 10 cifra
vector<int> prebrojCifre(const string& s) {
  vector<int> rez(10, 0);
  for (char c : s)
    rez[c - '0']++;
  return rez;
}

// proverava da li su dva stringa jednaka do na permutaciju karaktera
bool jednaki(string a, string b) {
  return prebrojCifre(a) == prebrojCifre(b);
}

int main() {
  // ucitavamo dva stringa
  string s, x;
  cin >> s >> x;
  int rez = 0;
  // analiziramo podstringove stringa s duzine jednake duzini stringa x
  for (int i = 0; i + x.length() - 1 < s.length(); i++)
    if (jednaki(s.substr(i, x.length()), x))
      rez++;
  cout << rez << endl;
  return 0;
}
```

Имплицитне подниске

Приметимо да се у претходном решењу непотребно у меморији ефективно граде све подниске, као и да се бројање карактера ниске `x` непотребно врши изнова за сваку подниску ниске `s`. Пошто се ниска `x` не мења, има смисла на самом почетку израчунати број појављивања цифара у `x` и тај низ користити све време.

Пребројавање карактера у делу ниске можемо издвојити у засебну функцију која уз ниску прима позицију од које почиње подниска и дужину те подниске.

Ове оптимизације убрзавају програм неколико пута (на пример, током поређења уместо израчунавања два низа бројача израчунавамо само један), али асимптотска сложеност и даље $O(S \cdot X)$.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

// prebrojava koliko se puta u delu stringa s (koji se sastoji samo od
// cifara) na pozicijama [start, start + n) javlja svaka od 10 cifra
vector<int> prebrojCifre(const string& s, int start, int n) {
```

```
  vector<int> rez(10, 0);
  for (int i = start; i < start + n; i++)
    rez[s[i] - '0']++;
  return rez;
}

int main() {
  string s, x;
  cin >> s >> x;
  // predstavljamo string x preko broja pojavljivanja svake cifre
  vector<int> brX = prebrojCifre(x, 0, x.length());
  // trazeni rezultat - broj pojavljivanja permutovanih podniski
  int rez = 0;
  // obradjujemo sve podstringove stringa s jednako dugacke kao x
  for (int pocetak = 0; pocetak + x.length() - 1 < s.length(); pocetak++) {
    // predstavljamo tekuci podstring preko broja pojavljivanja svake cifre
    vector<int> brS = prebrojCifre(s, pocetak, x.length());
    // proveravamo da li je tekuci podstring do na permutaciju jednak
    // stringu s
    if (brX == brS)
      rez++;
  }
  // ispisujemo rezultat
  cout << rez << endl;
}
```

Ако користимо класични низ уместо библиотечких колекција, тада можемо дефинисати функцију за поређење једнакости два низа. У језику C++ за поређење једнакости два класична низа можемо употребити и библиотечку функцију `equal`.

```
#include <iostream>
#include <string>

using namespace std;

// provera da li su dva niza jednaka
bool jednaki(int x[], int y[]) {
  for (int i = 0; i < 10; i++)
    if (x[i] != y[i])
      return false;
  return true;
}

int main() {
  // ucitavamo niske i racunamo njihovu duzinu
  string s, x;
  cin >> s >> x;
  int ds = s.length(), dx = x.length();

  // broj pojavljivanja svake cifre niske x
  int brX[10] = {0};
  for (int i = 0; i < dx; i++)
    brX[x[i]-'0']++;

  // trazeni rezultat - broj pojavljivanja permutovanih podniski
  int rez = 0;
  // obradjujemo sve podniske niske s jednako dugacke kao x
  for (int poc = 0; poc + dx - 1 < ds; poc++) {
    // broj pojavljivanja svake cifre unutar tekuce podniske niske s
```

```
    int brS[10] = {0};
    for (int i = 0; i < dx; i++)
      brS[s[poc + i] - '0']++;
    // ako su nizovi brojaca jednaki, uvecavamo rezulat
    if (jednaki(brS, brX))
      rez++;
  }

  // ispisujemo rezultat
  cout << rez << endl;

  return 0;
}
```

Инкременталност

Слично као у задатку Панграми, програм се може додатно значајно убрзати ако се искористи чињеница да се две узастопне подниске која се проверавају разликују само за почетни и крајњи карактер тј. да деле велики број својих карактера. Наредна подниска се од претходне добија уклањањем првог карактера те претходне подниске и додавањем наредног карактера ниске `s`. Стога можемо искористити принцип инкременталности и уместо да приликом преласка на наредну подниску број њених цифара рачунамо скроз из почетка, можемо их добити на основу претходног низа бројача тако што смањимо број појављивања првог карактера текуће подниске и увећамо број појављивања наредног карактера ниске `s`.

Пошто је у овом случају прелазак са претходне на наредну подниску операција која захтева константан број корака, укупна сложеност алгоритма се може грубо проценити са $O(S)$.

```
#include <iostream>
#include <algorithm>
#include <string>

using namespace std;

int main() {
  // ucitavamo niske i racunamo njihovu duzinu
  string s, x;
  cin >> s >> x;
  int ds = s.length(), dx = x.length();

  // izracunavamo broj pojavljivanja cifara unutar niske x i
  // pocetne podniske niske s (duzine jednake duzini niske x)
  int brS[10] = {0};
  int brX[10] = {0};
  for(int i = 0; i < dx; i++) {
    brS[s[i] - '0']++;
    brX[x[i] - '0']++;
  }

  // trazeni rezultat - broj pojavljivanja permutovanih podniski
  int rez = 0;
  // proveravamo pocetnu podnisku
  if (equal(brS, brS+10, brX))
    rez++;
  // analiziramo jedan po jedan preostali karakter niske s
  for (int i = dx; i < ds; i++) {
    // inkrementalno azuriramo broj pojavljivanja cifara
    // unutar podniske
    brS[s[i - dx] - '0']--;
    brS[s[i] - '0']++;
```

```
    // proveravamo tekucu podnisku
    if (equal(brS, brS+10, brX))
      rez++;
  }

  // ispisujemo konacan rezultat
  cout << rez << endl;

  return 0;
}
```

Мало читљивији програм можемо добити ако издвојимо неколико помоћних функција.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

// prebrojava koliko se puta u prefiksu stringa s duzine n (koji se
// sastoji samo od cifara) javlja svaka od 10 cifra
vector<int> prebrojCifre(const string& s, int n) {
  vector<int> rez(10, 0);
  for (int i = 0; i < n; i++)
    rez[s[i] - '0']++;
  return rez;
}

// stringu koji se sastoji samo od cifara i koji je predstavljen preko
// broja pojavljivanja svake cifre dodaje se nova cifra c
void dodajCifru(char c, vector<int>& brojCifara) {
  brojCifara[c - '0']++;
}

// stringu koji se sastoji samo od cifara i koji je predstavljen preko
// broja pojavljivanja svake cifre uklanja se cifra c
void ukloniCifru(char c, vector<int>& brojCifara) {
  brojCifara[c - '0']--;
}

int main() {
  // ucitavamo dva stringa
  string s, x;
  cin >> s >> x;
  // predstavljamo string x preko broja pojavljivanja svake cifre
  vector<int> brX = prebrojCifre(x, x.length());
  // predstavljamo pocetni podstring stringa s preko broja
  // pojavljivanja svake cifre
  vector<int> brS = prebrojCifre(s, x.length());
  // trazeni rezultat - broj pojavljivanja permutovanih podniski
  int rez = 0;
  // proveravamo da li je pocetni podstring stringa s do na
  // permutaciju jednak stringu x
  if (brX == brS)
    rez++;
  // analiziramo naredne podstringove, pomerajuci se za po jednu
  // poziciju nadesno
  for (int i = x.length(); i < s.length(); i++) {
    // uklanjamo prvu cifru tekuceg podstringa
```

```
    ukloniCifru(s[i - x.length()], brS);
    // dodajemo narednu cifru stringa s i tako dobijamo naredni podstring
    dodajCifru(s[i], brS);
    // proveravamo da li je on do na permutaciju jednak stringu x
    if (brX == brS)
      rez++;
  }
  cout << rez << endl;
}
```

Мана свих претходних решења је то што се поређење низа бројача увек врши изнова. Уместо експлицитног поређења низова бројача цифара након сваког њиховог ажурирања, можемо одржавати број једнаких парова бројача у текућа два низа и тај број инкрементално ажурирати. Два низа бројача су једнака ако и само ако су им вредности једнаке за сваку цифру тј. ако је број једнаких парова бројача једнак 10.

С обзиром на мали број бројача (само 10), приликом процене асимптотске сложености поређење низова бројача смо сматрали операцијом константне сложености. Због тога се на овај начин програм не убрзава асимптотски (сложеност и даље остаје $O(S)$), али се константни фактор смањује неколико пута.

```
#include <iostream>
#include <algorithm>
#include <string>

using namespace std;

int main() {
  // ucitavamo niske i racunamo njihovu duzinu
  string s, x;
  cin >> s >> x;
  int ds = s.length(), dx = x.length();

  // izracunavamo broj pojavljivanja cifara unutar niske x i
  // pocetne podniske niske s (duzine jednake duzini niske x)
  int brS[10] = {0};
  int brX[10] = {0};
  for(int i = 0; i < dx; i++) {
    brS[s[i] - '0']++;
    brX[x[i] - '0']++;
  }

  // trazeni rezultat - broj pojavljivanja permutovanih podniski
  int rez = 0;

  // broj cifara u X i tekucoj podniski od S koje se
  // pojavljuju jednak broj puta
  int brJednakih = 0;
  for (int i = 0; i < 10; i++)
    if (brS[i] == brX[i])
        brJednakih++;

  // proveravamo pocetnu podnisku
  if (brJednakih == 10)
    rez++;
  // analiziramo jedan po jedan preostali karakter niske s
  for (int i = dx; i < ds; i++) {
    // inkrementalno azuriramo broj pojavljivanja cifara unutar
    // podniske a takodje inkrementalno azuriramo i broj cifara koje
    // se pojavljuju jednak broj puta
    int cIzbac = s[i - dx] - '0';
    if (brS[cIzbac] == brX[cIzbac]) brJednakih--;
```

```
    brS[cIzbac]--;
    if (brS[cIzbac] == brX[cIzbac]) brJednakih++;

    int cUbac = s[i] - '0';
    if (brS[cUbac] == brX[cUbac]) brJednakih--;
    brS[cUbac]++;
    if (brS[cUbac] == brX[cUbac]) brJednakih++;

    // proveravamo tekucu podnisku
    if (brJednakih == 10)
      rez++;
  }

  // ispisujemo konacan rezultat
  cout << rez << endl;

  return 0;
}
```

Задатак: Сегмент највећег просека

Дат је низ a реалних бројева дужине n и природан број k. Написати програм којим се у низу a одређује позиција почетка сегмента (подниза узастопних елемената) дужине k са највећим просеком (ако више сегмената има исти просек, пријавити последњи од њих).

Улаз: У првој линији стандардног улаза налази се природан број k ($k \leq 5 \cdot 10^3$). У другој линији налази се природан број n ($n \leq 5 \cdot 10^5$). У следећих n линија налазе се по један реалан број (ти бројеви представљају редом елементе низа a).

Излаз: На стандарном излазу приказати позицију почетка последњег сегмента дужине k низа a чији је просек највећи (позиције у низу се броје од нуле).

Пример

Улаз	*Излаз*
3 5 1.0 5.0 8.0 2.0 7.0	2

Задатак: Панграми

Панграми су речи које садрже бар једно појављивање сваког слова абецеде или азбуке (слова се могу појављивати и више пута). Чувени панграм у енглеском језику је “the quick brown fox jumps over a lazy dog”. Напиши програм који проверава да ли се у датом тексту налази неки подтекст (низ узастопних карактера) дужине k који је панграм.

Улаз: Прва линија стандардног улаза садржи ниску састављену само од малих слова енглеске абецеде, дужине највише 10^5 карактера. Наредни ред садржи природан број k ($1 \leq k \leq 10^5$).

Излаз: На стандардни излаз исписати `da` ако у унетом тексту постоји панграм дужине k, односно `ne` у супротном.

Пример 1

Улаз	*Излаз*
xxxabcdefghijklmnopqrstuvwxyzxxx 26	da

Пример 2

Улаз	*Излаз*
xxxabcdefghijklmxxxnopqrstuvwxyzxxx 28	ne

Пример 3

Улаз	*Излаз*
xxxabcdefghijklmxxxnopqrstuvwxyzxxx 29	da

2.11.5 Инкременталност - вишедимензионални низови

Задатак: Суме трапеза

На основу квадратне матрице A_{nxn} формирати матрицу B_{nxn} такву да је $b_{i,j}$ једнак суми оних $a_{p,q}$ за које је $p \leq i$, а $p + q \leq i + j$. Ти елементи су смештени у правоугаоном трапезу, како је приказано на слици.

$a_{0,0}$	...	...	...	...	...	$a_{0,i+j}$	...
...	...	...	...	...	$a_{1,i+j-1}$	...	...
...	...	...	...	...	...	...	...
...	...	...	$a_{i-1,j+1}$	...	...	...	...
$a_{i,0}$	...	$a_{i,j}$	...	...	...	...	...
...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...

Слика 2.11: Суме трапеза

Улаз: Са стандардног улаза се учитава број n ($3 \leq n \leq 750$), а затим и квадратна матрица димензије nxn која садржи бројеве између 0 и 9.

Излаз: На стандардни излаз исписује се матрица сума трапезних области.

Пример

Улаз	*Излаз*
4 1 0 5 1 3 1 2 1 4 1 0 2 0 1 3 4	1 1 6 7 4 10 13 14 14 18 19 21 18 20 25 29

Решење

Груба сила

Формулација задатка сугерише једноставно, али неефикасно, решење које обично прво пада на памет: да за свако v и k израчунамо $B_{v,k}$ као збир елемената трапеза коме је (v, k) доње десно теме кроз које пролази крак трапеза. За то израчунавање се користе два помоћна циклуса. Спољни циклус пролази кроз врсте (од почетне врсте 0 па до текуће врсте v), док унутрашњи пролаи кроз колоне (од почетне колоне 0 па до крака трапеза или десне ивице матрице). За текућу врсту v', колону k' у којој се налази тачка на краку чије је доње теме у тачки (v, k) можемо једноставно израчунати на основу чињенице да је за све тачке на краку збир врсте и колоне константан, па је $v + k = v' + k'$ тј. $k' = v + k - v'$. Додатно мора да важи да је $k' < n$. Укупна сложеност овог поступка је $O(n^4)$.

```
#include <iostream>
using namespace std;

const int MAX = 750;
int A[MAX][MAX], B[MAX][MAX];


// ucitava kvadratnu matricu sa standardnog ulaza - ucitava se
// prvo dimenzija matrice, a zatim elementi matrice red po red
```

```
void ucitaj(int A[][MAX], int& n) {
  cin >> n;
  for (int v = 0; v < n; v++)
    for (int k = 0; k < n; k++)
      cin >> A[v][k];
}

// formira trazenu matricu zbirova trapeza
void saberiTrapeze(int A[][MAX], int B[][MAX], int n) {
  for (int v = 0; v < n; v++)
    for (int k = 0; k < n; k++) {
      // sabiramo trapez cije je donje desno teme (v, k)
      B[v][k] = 0;
      for (int vv = 0; vv <= v; vv++) {
        for (int kk = 0; kk <= min(v + k - vv, n - 1); kk++)
          B[v][k] += A[vv][kk];
      }
    }
}

// ispisuje kvadratnu matricu na standardni izlaz
void ispisi(int A[][MAX], int n) {
  for (int v = 0; v < n; v++) {
    for (int k = 0; k < n; k++)
      cout << A[v][k] << " ";
    cout << endl;
  }
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  ucitaj(A, n);
  saberiTrapeze(A, B, n);
  ispisi(B, n);
  return 0;
}
```

Префиксни збирови врста

Свака сума трапеза се може разложити на суму својих врста, а свака сума елемената неке врсте трапеза заправо је нека парцијална сума те врсте унутар полазне матрице. Стога алгоритам можемо убразати ако унапред израчунамо све парцијалне суме врста полазне матрице. Пошто нам након израчунавања парцијалних сума елементи полазне матрице више нису потребни, за смештање парцијалних сума можемо употребити полазну матрицу (да не бисмо ангажовали додатну меморију). Дакле, трансформишемо матрицу A у матрицу A' тако да је сваки елемент $A'_{v,k} = \sum_{i=0}^{k} A_{v,i}$. Збирове префикса можемо израчунати инкрементално, исто као у задатку Префикс највећег збира. Сложеног тог поступка је $O(n)$ за сваку врсту тј. укупно $O(n^2)$ за свих n врста.

Након израчунавања префиксних сума, суме трапеза чије је доње десно теме у врсти v израчунавамо сабирањем v префиксних сума, па је сложеност израчунавања сума трапеза у врсти v једнака $O(v^2)$. Пошто има укупно n врста, укупна сложеност овог решења је $O(n^3)$. Приметимо да је ово решење практично исто као оно грубом силом, осим што је избегнуто да се унутрашња петља на потпуно идентичан начин понавља више пута.

```
#include <iostream>
#include <algorithm>

using namespace std;
const int MAX = 750;
```

```
int A[MAX][MAX],B[MAX][MAX];

void saberiTrapeze(int A[][MAX], int B[][MAX], int n) {
  // izracunavamo parcijalne zbirove vrsta
  for (int v = 0; v < n; v++)
    for (int k = 1; k < n; k++)
      A[v][k] += A[v][k-1];

  // izracunavamo sume trapeza
  for (int v = 0; v < n; v++)
    for (int k = 0; k < n; k++) {
      // sabiramo odgovarajuce parcijalne zbirove vrsta (zakljucno sa vrstom v)
      B[v][k] = 0;
      for(int vv = 0; vv <= v; vv++)
        B[v][k] += A[vv][min(v + k - vv, n - 1)];
    }
}

void ispisi(int A[][MAX], int n) {
  for (int v = 0; v < n; v++) {
    for (int k = 0; k < n; k++)
      cout << A[v][k] << " ";
    cout << endl;
  }
}

void ucitaj(int A[][MAX], int& n) {
  cin >> n;
  for (int v = 0; v < n; v++)
    for (int k = 0; k < n; k++)
      cin >> A[v][k];
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  ucitaj(A, n);
  saberiTrapeze(A, B, n);
  ispisi(B, n);
  return 0;
}
```

Инкрементално израчунавање

У следећем алгоритму избегавамо понављање истих сабирања тако што користимо већ израчунате збирове мањих трапеза (слично као у задатку Минимуми правоугаоника). Размотримо трапез чији збир елемената треба уписати на позицију (v, k). Претпоставимо, за почетак да се (v, k) не налази на левој, горњој, нити десној ивици матрице (тј. да је $v \neq 0$ и $k \neq 0$ и $k \neq n - 1$). Сви елементи који су на наредној слици означени најсветлијом и средње светлом сивом бојом део су збира $B_{v,k-1}$. Слично, сви елементи означени најтамнијом и најсветлијом сивом бојом део су збира $B_{v-1,k+1}$. Сви елементи који су означени најсветлијом сивом бојом део су збира $B_{v-1,k}$.

$a_{0,0}$	...	...	...	...	...	...	$a_{0,v+k}$	...
...	...	...	...	...	...	$a_{1,v+k-1}$	...	...
...	...	...	...	...	...	...	...	...
...	...	...	$a_{v-1,k}$	$a_{v-1,k+1}$	...	...	...	...
$a_{v,0}$	...	$a_{v,k-1}$	$a_{v,k}$	...	...	...	...	...
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Слика 2.12: Суме трапеза

Пошто се у збиру $B_{v,k-1}+B_{v-1,k+1}+A_{v,k}$ два пута појављују елементи њиховог пресека чији је збир $B_{v-1,k}$, важи да је

$$B_{v,k} = B_{v,k-1} + B_{v-1,k+1} - B_{v-1,k} + A_{v,k}$$

За остале елементе слично се добија:

$$\begin{aligned} B_{0,0} &= A_{0,0}, \\ B_{0,k} &= B_{0,k-1} + A_{0,k}, \quad \text{за } k > 0, \\ B_{v,0} &= B_{v-1,1} + A_{v,0}, \quad \text{за } 0 < v < n, \\ B_{v,n-1} &= B_{v,n-2} + A_{v,n-1}, \quad \text{за } 0 < v < n. \end{aligned}$$

Коришћењем ових формула, ако је израчунавање по врстама слева надесно, проблем се може решити у две петље и без коришћења помоћне матрице B, у сложености $O(n^2)$.

```
#include <iostream>
#include <algorithm>

using namespace std;

const int MAX = 750;

int A[MAX][MAX],B[MAX][MAX];

// formira trazenu matricu minimuma
void saberiTrapeze(int A[][MAX], int B[][MAX], int n) {
  for (int v = 0; v < n; v++)
    for (int k = 0; k < n; k++) {
      B[v][k] = A[v][k];
      // izracunava B[0][0]
      if (v == 0 && k == 0)
        continue;
       // izracunava elemente prve vrste
      if (v == 0) {
        B[0][k] += B[0][k - 1];
        continue;
      }
      // izracunava elemente prve kolone
      if (k == 0) {
        B[v][0] += B[v - 1][1];
        continue;
      }
      // izracunava elemente poslednje kolone
      if (k == n - 1) {
```

```
        B[v][k] += B[v][k - 1];
        continue;
      }
      // izracunava ostale elemente
      B[v][k] += B[v][k-1] + B[v-1][k+1] - B[v-1][k];
    }
}

// ispisuje kvadratnu matricu na standardni izlaz
void ispisi(int A[][MAX], int n) {
  for (int v = 0; v < n; v++) {
    for (int k = 0; k < n; k++)
      cout << A[v][k] << " ";
    cout << endl;
  }
}

// ucitava kvadratnu matricu sa standardnog ulaza - ucitava se
// prvo dimenzija matrice, a zatim elementi matrice red po red
void ucitaj(int A[][MAX], int& n) {
  cin >> n;
  for (int v = 0; v < n; v++)
    for (int k = 0; k < n; k++)
      cin >> A[v][k];
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  ucitaj(A, n);
  saberiTrapeze(A, B, n);
  ispisi(B, n);
  return 0;
}
```

Задатак: Минимуми правоугаоника

За дату матрицу $A_{m\times n}$ одредити матрицу која се добија када се сваки елемент $a_{i,j}$ замени минималним елементом подматрице $A'_{i\times j}$, чије је горње лево теме на позицији $(0,0)$, а доње десно на позицији (i,j).

Улаз: Са стандардног улаза се учитавају бројеви m и n ($3 \leq m,n \leq 500$), а затим и правоугаона матрица димензије $m \times n$ која садржи бројеве између 0 и 1000000.

Излаз: На стандардни излаз исписује се матрица минимума правоугаоних области.

Пример

Улаз	*Излаз*
3 4 5 1 2 3 4 3 2 0 3 4 1 2	5 1 1 1 4 1 1 0 3 1 1 0

2.11.6 Инкременталност - суфикси до сваке позиције у низу

Задатак: Број растућих сегмената

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Индуктивна конструкција

Најлепше решење можемо добити ако употребимо инкрементално проширивање сегмената једним по једним елементом здесна. Укупан број растућих сегмената у низу можемо добити тако што за сваку позицију низа израчунамо број R растућих сегмената који се завршавају на тој позицији и тако добијене бројеве саберемо.

На позицији 0 се не завршава ни један растући сегмент (јер они по дефиницији морају да буду бар двочлани). Дакле, важи $R_0 = 0$. Ако знамо број растућих сегмената који се завршавају на позицији j, тада веома једноставно можемо израчунати број растућих сегмената који се завршавају на позицији $j + 1$.

- Ако је $a_{j+1} > a_j$, онда се сваки растући сегмент који се завршава на позицији j може проширити елементом a_{j+1} и тако добити нови растући сегмент. Додатно, растући сегмент је и $[a_j, a_{j+1}]$, тако да је укупан број сегмената који се завршавају на позицији $j + 1$ за један већи од укупног броја растућих сегмената који се завршавају на позицији j и важи $R_{j+1} = R_j + 1$.
- Ако је $a_{j+1} \leq a_j$ онда се на позицији j не завршава ни један растући сегмент тј. важи $R_{j+1} = 0$.

Број растућих сегмената за сваки десни крај и њихов укупан број одређујемо једним проласком кроз низ, у сложености $O(n)$. Пошто памтимо само да узастопна члана низа, меморијска сложеност је $O(1)$.

```
int n;
cin >> n;
int prethodni;
cin >> prethodni;
// ukupan broj rastucih serija
int ukupanBrojRastucih = 0;
// broj rastucih koji se zavrsavaju na tekucoj poziciji
int brojRastucih = 0;
for(int i = 1; i < n; i++) {
  int tekuci;
  cin >> tekuci;
  if (tekuci > prethodni) {
    // tekuci element produzava sve rastuce segmente koji su se zavrsili na
    // prethodnoj poziciji i dodaje jos jedan nov dvoclan rastuci segment
    brojRastucih++;
    // dodajemo broj rastucih koji se zavrsavaju na poziciji i na
    // ukupan broj rastucih segmenata
    ukupanBrojRastucih += brojRastucih;
  } else {
    // na tekucoj poziciji se ne zavrsava ni jedna rastuca serija
    brojRastucih = 0;
  }
  prethodni = tekuci;
}
cout << ukupanBrojRastucih << endl;
```

Задатак: Најдужа серија победа

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Серија сјајних партија

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Број сегмената исте парности

Напиши програм који одређује колико у датом низу постоји непразних сегмената (поднизова узастопних елемената) чији су сви елементи исте парности.

Улаз: Са стандардног улаза се учитава дужина низа n ($1 \leq n \leq 50000$), а затим у наредних n редова по један природан број који представља члан низа.

Излаз: На стандардни излаз исписати тражени број сегмената.

Пример

Улаз	*Излаз*
5 1 3 5 2 4	9

Објашњење

То су сегменти 1, 1 3, 1 3 5, 3, 3 5, 5, 2, 2 4 и 4.

Задатак: Број сегмената са различитим елементима

Дат је низ природних бројева дужине n. Написати програм којим се одређује колико има сегмената у датом низу чији су сви елементи различити. Сегмент низа чине узастопни елементи низа (њих бар 2).

Улаз: Прва линија стандардног улаза садржи природан број n ($2 \leq n \leq 50000$), број елемената низа. У свакој од n наредних линија стандардног улаза, налази по један члан низа.

Излаз: На стандардном излазу приказати у једној линији број сегмената датог низа чији су сви елементи различити.

Пример

Улаз	*Излаз*
5 1 2 2 3 6	4

Решење

Максимални сегменти за фиксирани крај

Једно прилично елегантно решење се заснива на томе да за сваку позицију анализирамо све сегменте којима је крај управо на тој позицији, а којима су сви елементи различити. Приметимо да ако неки сегмент има то својство, онда и сви сегменти иза њега (његови суфикси) такође имају то својство, док ако неки сегмент нема то својство, онда то својство нема ни један сегмент испред њега (његов префикс). Зато је довољно да пронађемо најдужи могући сегмент који се завршава на позицији $kraj$ и којем су сви елементи различити. Ако је то сегмент $[pocetak, kraj]$ онда ће такви бити и сегменти $[pocetak+1, kraj]$, ..., $[kraj-1, kraj]$. Њих је укупно $kraj - pocetak$.

Остаје питање како одредити позицију $pocetak$. Претпоставимо да већ знамо решење за претходну позицију краја, тј. претпоставимо да знамо да је $[pocetak, kraj-1]$, најдужи сегмент који има све различите елементе и који се завршава на позицији $kraj - 1$ (у старту можемо и $pocetak$ иницијализовати на нулу, а $kraj$ на јединицу, знајући да $[0, 0]$ не садржи дупликате и најдужи је такав који се завршава на позицији нула). Ако се елемент на позицији $kraj$ не садржи у том сегменту, онда је сегмент $[pocetak, kraj]$ наш тражени. Ако се садржи, онда је он сигурно једини дупликат у сегменту $[pocetak, kraj]$. Ако претпоставимо да се елемент на позицији $kraj$ у том сегменту јавља и на позицији p, тада је сегмент који тражимо $[p+1, kraj]$, зато што сви сегменти који почињу од позиције $pocetak$, па све до позиције p садрже исти тај дупликат. Сегмент $[p+1, kraj]$ не садржи дупликате и најдужи је такав сегмент, тако да у тој ситуацији $pocetak$ треба поставити на вредност $p + 1$.

На крају, остаје питање како утврдити да ли се a_{kraj} јавља у сегменту $[pocetak, kraj - 1]$ и ако се јавља, како одредити позицију p на којој се јавља. Директно решење подразумева линеарну претрагу сегмента приликом сваког проширења низа, што би знатно деградирало сложеност целог алгоритма. Боље решење је да се чува асоцијативни низ (мапа, речник) у којем се елементи низа из сегмента позиција $[pocetak, kraj - 1]$ пресликавају у њихове позиције. Тада се једноставном претрагом мапе тј. речника (чија је сложеност константна или највише логаритамска) утврђује да ли се нови крајњи елемент јавља у претходном сегменту

и на исти начин се одређује и његова позиција. Рецимо и да се приликом померања почетка на позицију $p+1$ сегмент скраћује, што треба да се ослика и у мапи - зато је тада потребно из мапе уклонити све елементе који се јављају у низу, на позицијама од $pocetak$, па закључно са p.

Приметимо одређену сличност овог алгоритма са оним приказаним у задатку Најкраћа подниска која садржи све дате карактере, где смо такође анализирали сегменте са фиксираним крајем и ослањали се на познато решење у којем је крај био једну позицију пре текуће.

И почетак и крај се само увећавају (никада се не умањују), што значи да се укупно изврши највише $O(n)$ корака у којима се врши претрага мапе, што значи да је сложеност највише $O(n \log n)$ – ако се користи мапа заснована на хеширању, сложеност је $O(n)$, уз могуће мало веће заузеће меморије.

```
#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

int main() {
  // ucitavamo dati niz brojeva
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ukupan broj segmenata niza ciji su svi elementi razliciti
  int broj = 0;

  // za svaku poziciju kraj zelimo da pronadjemo najduzi segment
  // oblika [pocetak, kraj] koji ima sve razlicite elemente

  // za svaki element u tekucem segmentu [pocetak, kraj] pamtimo
  // poziciju na kojoj se pojavljuje
  unordered_map<int, int> prethodno_pojavljivanje;

  int pocetak = 0;
  for (int kraj = 0; kraj < n; kraj++) {
    // karakter a[kraj] se vec javio u segmentu [pocetak, kraj-1]?
    if (prethodno_pojavljivanje.find(a[kraj]) != prethodno_pojavljivanje.end()) {
      // nijedan segment koji se zavrsava na poziciji kraj, a pocinje
      // pre ranijeg pojavljivanja elementa a[kraj] ne moze da ima sve
      // razlicite elemente, pa zato razmatramo samo segmente koji se
      // zavrsavaju na poziciji kraj i pocinju iza pozicije tog
      // prethodnog pojavljivanja - najduzi takav pocinje na prvoj
      // poziciji iza te pozicije
      int novi_pocetak = prethodno_pojavljivanje[a[kraj]] + 1;
      // brisemo iz segmenta sve elemente od starog do ispred novog pocetka
      // i mapu uskladjujemo sa time
      for (int i = pocetak; i < novi_pocetak; i++)
        prethodno_pojavljivanje.erase(a[i]);
      // pomeramo pocetak
      pocetak = novi_pocetak;
    }
    // prosirujemo segment elementom a[kraj], pa pamtimo poziciju
    // njegovog pojavljivanja
    prethodno_pojavljivanje[a[kraj]] = kraj;

    // [pocetak, kraj] sadrzi sve razlicite elemente i on je najduzi
    // takav koji se zavrsava na poziciji kraj
```

```
    // sigurno su takvi i [pocetak+1, kraj], ..., [kraj-1, kraj]
    // njih ima (kraj - pocetak) i taj broj dodajemo na ukupan broj
    // trazenih segmenata
    broj += kraj - pocetak;
  }

  // ispisujemo ukupan broj pronadjenih segmenata
  cout << broj << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Максимални збир сегмента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Максимални суфикси

С обзиром на то да се суфикси првих i елемената низа могу анализирати инкрементално (суфикси $i+1$ елемената низа се добијају проширивањем суфикса i елемената низа додатним елементом), проблем анализе свих сегмената је пожељно свести на проблем анализе суфикса. У овом конкретном проблему, можемо приметити да је максимални збир сегмента уједно и максимални збир суфикса који се завршава на позицији на којој се тај сегмент завршава. Стога се задатак може свести на то да се за сваку позицију у низу одреди максимални суфикс, а да се онда међу максималним суфиксима за сваку позицију пронађе онај који је глобално максимални.

Једини суфикс првих нула елемената низа је празан и његов збир је по дефиницији 0. Сви суфикси првих $i+1$ елемената низа, изузев празног суфикса добијају се додавањем елемента на позицији i на крај неког суфикса првих i елемената низа. Међу непразним суфиксима највећи збир има онај који је добијен додавањем последњег елемента управо на суфикс првих i елемената низа који има максимални збир. Од њега једино може бити већи збир празног суфикса (и то када се након проширивања последњим елементом добије негативни збир). Ако вредности максималног збира суфикса памтимо у низу, тада низ лако попуњавамо на основу веза $S_0 = 0$ и $S_{i+1} = \max(S_i + a_i, 0)$, где је са S_i обележена вредност максималног збира суфикса првих i елемената низа a.

На крају налазимо максимум низа S_i.

Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности S_i.

```
i    a_{i+1}   S_i
0              0
1    -2        0 = max(0+(-2), 0)
2     3        3 = max(0+3, 0)
3     2        5 = max(3+2, 0)
4    -3        2 = max(5+(-3), 0)
5    -3        0 = max(2+(-3), 0)
6    -2        0 = max(0+(-2), 0)
7     4        4 = max(0+4, 0)
8     5        9 = max(4+5, 0)
9    -8        1 = max(9+(-8), 0)
10    3        4 = max(1+3, 0)
```

Максимална вредност у колони S_i је 9.

Пошто користимо низ максималних збирова суфикса, меморијска сложеност је $O(n)$. Низ попуњавамо елемент по елемент, инкрементално, у једном пролазу за шта је довољно n корака, а затим максимум налазимо у новом пролазу тј. у нових n корака. Укупна сложеност је, дакле, линеарна тј. $O(n)$.

```
// maksimalni sufiks prvih i elemenata niza
vector<int> maxSufiks(n+1);
maxSufiks[0] = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  maxSufiks[i+1] = max(maxSufiks[i] + x, 0);
}

// maksimalni segment je maksimalni od svih maksimalnih sufiksa
cout << *max_element(begin(maxSufiks), end(maxSufiks)) << endl;
```

Каданов алгоритам

Максималне вредности збирова суфикса не морамо да памтимо у низу, ако њихов максимум одређујемо истовремено са одређивањем вредности максималних збирова суфикса. Овај алгоритам познат је под именом *Каданов алгоритам*.

Један начин да се дође до тог алгоритма је следећи. Покушавамо да алгоритам заснујемо на индуктивној конструкцији.

- За празан низ, једини сегмент је празан и његов је збир нула (то је уједно највећи збир који се може добити).
- Сматрамо да умемо да проблем решимо за произвољан низ дужине n и на основу тога покушавамо да решимо задатак за низ дужине $n + 1$ (полазни низ проширен једним додатним елементом).

  Сегмент највећег збира у проширеном низу се или цео садржи у полазном низу дужине n или чини суфикс проширеног низа, тј. завршава се на последњој позицији (укључујући и могућност да је ту и празан сегмент).

  На основу индуктивне хипотезе знамо да израчунамо највећи збир сегмента низа дужине n и потребно је да још одредимо максимални збир суфкса проширеног низа. Један начин да се то уради је да приликом сваког проширења низа изнова анализирамо све сегменте који се завршавају на текућој позицији, али чак иако то радимо инкрементално (кренувши од празног суфикса, па додајући уназад један по један елемент) највише што можемо добити је алгоритам квадратне сложености (пробајте да се уверите да је то заиста тако). Кључни увид је то да највећи збир суфикса који се завршава на текућој позицији можемо инкрементално израчунати знајући највећи збир суфикса низа пре проширења. Највећи збир неког непразног суфикса који се завршава на текућој позицији је збир текућег елемента низа и највећег збира неког суфикса који се завршава на претходној позицији. Од њега може бити повољнији само празан суфикс (и то само ако је претходни збир негативан).

Дакле, ако са S_i обележимо максимални збир неког суфикса првих i елемената низа, а са M_i максимални збир неког сегмента прих i елемената низа, важи да је $M_0 = S_0 = 0$, да је $S_{i+1} = \max(S_i + a_i, 0)$ и $M_{i+1} = \max(M_i, S_{i+1})$.

Имплементацију можемо направити итеративним алгоритмом коме је инваријанта да у сваком кораку петље знамо ове две вредности (максимум сегмента и максимум суфикса).

Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности S_i и M_i.

```
i    a_{i+1}   S_i                    M_i
0              0                      0
1    -2        0 = max(0+(-2), 0)     0 = max(0, 0)
2     3        3 = max(0+3, 0)        3 = max(0, 3)
3     2        5 = max(3+2, 0)        5 = max(3, 5)
4    -3        2 = max(5+(-3), 0)     5 = max(5, 2)
5    -3        0 = max(2+(-3), 0)     5 = max(5, 0)
6    -2        0 = max(0+(-2), 0)     5 = max(5, 0)
7     4        4 = max(0+4, 0)        5 = max(5, 4)
8     5        9 = max(4+5, 0)        9 = max(5, 9)
9    -8        1 = max(9+(-8), 0)     9 = max(9, 1)
10    3        4 = max(1+3, 0)        9 = max(9, 4)
```

Пошто елементе учитавамо један по један и не памтимо их истовремено, меморијска сложеност је $O(1)$. Максимални збир сегмента и суфикса инкрементално израчунавамо једним проласком кроз задате елементе и временска сложеност је линеарна тј. $O(n)$.

Приметимо да смо у претходном разматрању проширили индуктивну хипотезу претпостављајући да поред тражене вредности тј. максимума неког сегмента првих n елемената низа знамо додатно и вредност максималног суфикса првих n елемената низа.

```
int maxSufiks = 0, max = maxSufiks;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  maxSufiks += x;
  if (maxSufiks < 0)
    maxSufiks = 0;
  if (maxSufiks > max)
    max = maxSufiks;
}
cout << max << endl;
```

Види другачија решења овог задатка.

2.12 Збирови префикса и разлике суседних елемената низа

Ако је дат низ елемената збир свих елемената у интервалу позиција $[a, b]$ се може израчунати као разлика између збира свих елемената у интервалу позиција $[0, b]$ и збира елемената у интервалу позиција $[0, a - 1]$.

На пример, размотримо како да израчунамо збир елемената на позицијама из интервала [3, 5] (тј. на позицијама 3, 4 и 5) у низу 4, 2, 3, 1, 5, 6, 9, 2. На тим позицијама се налазе елементи 1, 5 и 6 и збир им је $1 + 5 + 6 = 12$. Збир свих елемената из интевала позиција $[0, 5]$ је $4 + 2 + 3 + 1 + 5 + 6 = 21$, док је збир свих елемената из интервала позиција $[0, 2]$ једнак $4 + 2 + 3 = 9$. Разлика $21 - 9$ управо је једнака 12.

Ова наизглед веома једноставна особина сабирања може значајно помоћи убрзању разних алгоритама у којима су нам потребни збирови узастопних елемената низа. Наиме, ако знамо *збирове свих префикса* низа тј. збирове на свим интервалима $[0, k)$, за $k = 0$ до n (а њих можемо израчунати током фазе претпроцесирања, инкрементално, у линеарној сложености), тада у константној сложености (једним одузимањем) можемо израчунати збир произвољног сегмента низа.

У језику C++ парцијалне збирове је могуће израчунати и коришћењем библиотечке функције `partial_sum`, која, наравно, ради у линеарној сложености. Функцији се прослеђују два итератора на део низа који се сабира, као и итератор на почетак низа у који се смештају резултати (пошто се унапред зна колико ће елемената бити, тај низ се унапред алоцира).

Дакле, од датог низа, низ збирова префикса можемо израчунати у линеарној сложености, али важи и обратно. Од низа префикса, у линеарној сложености можемо израчунати елементе оригиналног низа. Важи чак и јаче тврђење од тога, јер сваки конкретни елемент низа можемо наћи у константној сложености, одузимањем два суседна збира префикса. Дакле, прелазак са низа на збирове његових префикса можемо сматрати променом репрезентације (нема смисла чувати и једно и друго истовремено у меморији).

Приметимо огромну сличност са интегралним и диференцијалним рачуном. Израчунавање збирова префикса одговара одређеном интегрaљењу, разлика збирова префикса одговара Њутн-Лајбницовој формули, док израчунавање разлике суседних елемената одговара диференцирању. Интеграљење и диференцирање су међусобно инверзне операције.

Дуалан приступ збировима префикса је промена репрезентације у којој уместо низа чувамо разлике суседних елемената. Повратак на оргинални низ се онда може извршити у линеарној сложености тако што израчунамо збирове префикса низа разлика. Ова репрезентација нам омогућава да веома ефикасно мењамо сегменте низа тако што све елементе из неког задатог сегмента увећамо или умањимо за неку фиксну вредност.

Задатак: Аритметички троугао

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Разлика два збира

Постоји веома елегантан начин да се израчуна тражени збир. Лако је уочити да је збир елемената k-тог реда једнак разлици збира свих елемената троугла коме је основица k-ти ред и збира свих елемената троугла коме је основица $(k-1)$-ви ред. Ако са s_n означимо збир бројева $1+2+\ldots+n$, збир вредности k-тог реда добијамо као $s_{k^2} - s_{(k-1)^2}$, јер се k-ти ред завршава са k^2, а $(k-1)$-ви са $(k-1)^2$. Збир $1+2+\ldots+n$ једнак је $\frac{n\cdot(n+1)}{2}$ (ова формула се приписује Гаусу и већ смо је дискутовали у задатку Спортске припреме).

У овом решењу треба обратити пажњу на случај у којем је тражен збир реда такав да се може исправно репрезентовати одабраним целобројним типом, док збир свих елемената троугла (збир тог и свих претходних редова) прекорачује опсег тог типа – с обзиром на ограничења дата у задатку, то овде није случај.

Као што ћемо видети, идеја израчунавања збира од $[a,b]$ као разлике између збира од $[0,b]$ и збира $[0,a-1]$ је изузетно значајна и јако се често користи.

```
#include <iostream>
using namespace std;

// suma brojeva 1, 2, ..., n
long long gaus(long long n) {
        return n * (n + 1) / 2;
}

int main() {
  long long k;
  cin >> k;
  long long zbirRedaTrougla = gaus(k * k) - gaus((k - 1) * (k - 1));
  cout << zbirRedaTrougla << endl;
  return 0;
}
```

Задатак: Збирови сегмената

Позната је зарада једног предузећа током одређеног броја дана. Напиши програм који омогућава кориснику да израчунава укупну зараду предузећа у временским периодима одређеним почетним и крајњим даном.

Улаз: Са стандардног улаза се уноси број дана n ($1 \leq n \leq 100000$), а затим у наредном реду n целих бројева између 0 и 100, раздвојених са по једним размаком, који представљају зараде током n дана. Након тога се уноси број упита m ($1 \leq m \leq 100000$) и у наредних m редова се уносе временски периоди одређени редним бројем почетног дана a и крајњег дана b ($0 \leq a \leq b < n$).

Излаз: На стандардни излаз исписати m целих бројева који представљају укупне зараде у сваком од m периода.

Пример

Улаз	*Излаз*
5 1 2 3 4 5 3 0 4 1 3 2 2	15 9 3

Задатак: Максимални збир сегмента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Збирови префикса

Један алгоритам којим можемо ефикасно решити овај задатак се заснива на коришћењу збирова префикса. Сличну техника је описана, на пример, у задатку Збирови сегмената. Ако знамо збир сваког префикса низа, онда збир сваког сегмента можемо добити као разлику збирова два префикса. Збир елемената сегмента $[l, d]$ једнак је разлици збира елемената сегмента $[0, d + 1)$ и збира елемената сегмента $[0, l)$, тј. важи да је

$$\sum_{i=l}^{d} a_i = \sum_{i=0}^{d} a_i - \sum_{i=0}^{l-1} a_i$$

Рачунамо да је збир празног сегмента $[0, 0)$ по дефиницији једнак нули.

За сваку позицију у низу одређујемо максимални збир суфикса који се на тој позицији завршава. Највећи од свих максималних збирова суфикса је највећи збир неког сегмента у низу (јер је сваки сегмент заправо суфикс дела низа до оне позиције на којој се тај сегмент завршава).

Користимо индуктивно-рекурзивну конструкцију и низ проширујемо једним по једним елементом. Крећемо од празног низа чији је максимални збир сегмента једнак нули. Приликом сваког проширивања низа новим елементом, претпостављамо да знамо максимум збирова сегмента у непроширеном делу низа и да израчунамо максимални збир суфикса проширеног низа. Максимум збирова сегмента у проширеном низу је већи од та два броја.

Максимални збир суфикса проширеног низа, на основу разлагања на збирове префикса, добија се као разлика збира целог проширеног низа (тј. збира префикса до текуће позиције) и збира неког префикса непроширеног низа (празан суфикс не морамо анализирати, јер је празан сегмент већ обрађен у склопу иницијализације). Пошто је умањеник константан, да бисмо максимизовали разлику потребно да знамо најмањи могући умањилац, тј. да знамо најмањи збир префикса који се завршава на некој позицији испред текуће. И текући и минимални збир префикса можемо одржавати инкрементално. Инкрементално израчунавање збирова префикса већ је описано у задатку Префикс највећег збира.

Када год низ проширимо неким елементом, збир префикса увећавамо за тај елемент, поредимо га са дотадашњим минималним збиром префикса и ако је мањи, ажурирамо минимални збир префикса. Наравно, одржавамо и глобални максимални збир сегмента који ажурирамо сваки пут када наиђемо на суфикс чији је збир већи од дотадашњег максимума.

Приметимо да је у овом решењу је индуктивна хипотеза појачана и претпостављамо да поред сегмента највећег збира у обрађеном делу низа умемо да одредимо и минимални збир префикса обрађеног дела низа.

Сложеност овог решења је $O(n)$, па је ово решење оптималне сложености.

```
int zbir_prefiksa = 0;
int min_zbir_prefiksa = zbir_prefiksa;
int max = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  zbir_prefiksa += x;
  int zbir_segmenta = zbir_prefiksa - min_zbir_prefiksa;
  if (zbir_segmenta > max)
    max = zbir_segmenta;
  if (zbir_prefiksa < min_zbir_prefiksa)
    min_zbir_prefiksa = zbir_prefiksa;
}
cout << max << endl;
```

Види другачија решења овог задатка.

Задатак: Најкраћи сегмент збира бар K

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Збирови префикса

Уместо инкременталности, можемо применити технику збирова префикса (слично као у задатку Збирови сегмената). Ако претпоставимо да су познати збирови z_i првих i елемената низа, онда је збир сегмента $[i, j]$ једнак $z_{j+1} - z_i$. Обрађујемо један по један елемент полазног низа, увећавамо збир префикса (важи да је $z_0 = 0$ и $z_{i+1} = z_i + a_i$) и затим анализирамо све збирове z_j за $0 \leq j \leq i$ и проверавамо да ли је $z_{i+1} - z_j$ (тј. збир сегмента $[j, i]$) веће од K и ако јесте, израчунавамо дужину $i - j + 1$ сегмента $[j, i]$ и ажурирамо минималну дужину ако је то потребно.

Унутрашња петља се извршава редом 1, 2, па све до n пута, па је временска сложеност $O(n^2)$.

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
  int k;
  cin >> k;
  int n;
  cin >> n;
  // duzina najkraceg segmenata (+beskonacno na pocetku)
  int minBrojElemenata = n + 1;
  // zbir prefiksa [0, i]
  int zbir = 0;
  // niz svih zbirova prefiksa -- na poziciji m je zbir prvih m
  // elemenata niza
  vector<int> zbirovi;
  zbirovi.push_back(zbir);
  // analiziramo sve leve krajeve segmenta
  for (int i = 0; i < n; i++) {
    // ucitavamo element x = a[i] i uvecavamo zbir prefiksa
    int x; cin >> x;
    zbir += x;
    // analiziramo sve segmente [j, i]
    for (int j = 0; j < zbirovi.size(); j++)
      // zbir segmenta [j, i] = zbirovi[i+1] - zbirovi[j]
      if (zbir - zbirovi[j] >= k)
        minBrojElemenata = min(minBrojElemenata, i - j + 1);

    // izracunati zbir prvih i+1 clanova niza ubacujemo u niz zbirova
    zbirovi.push_back(zbir);
  }

  // ispisujemo rezultat
  if (minBrojElemenata <= n)
    cout << minBrojElemenata << endl;
  else
    cout << "-" << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Сегмент датог збира у низу целих бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Сегмент датог збира у низу природних бројева

У датом низу позитивних природних бројева наћи све сегменте (њихов почетак и крај) чији је збир једнак датом позитивном броју (бројање позиција почиње од нуле).

Улаз: У првој линији стандардног улаза налази се задати позитиван природни број z који представља дати збир $0 < z < 10^6$, у другој број елемената низа, N ($2 \leq N \leq 50000$), а затим, у свакој од наредних N линија стандардног улаза, по један елемент низа (позитиван природни број мањи од 200).

Излаз: У свакој линији стандардног излаза исписују се два броја (цели бројеви) одвојена празнином, који представљају индексе почетка и краја сегмента (бројано од нуле). Ако постоји више тражених сегмената њихове индексе исписати сортирано на основу левог краја.

Пример

Улаз	*Излаз*
125 10 60 40 25 50 50 100 25 35 30 35	0 2 2 4 5 6 6 9

Задатак: Сегменти чији је збир дељив са k

Дат је низ a природних бројева дужине n и природан број k. Пера треба да изабере сегмент низа a (непразан подниз узастопних елемената) чија је сума дељива са k. Написати програм којим се одређује на колико начина то Пера може да уради.

Улаз: У првој линији стандардног улаза налази се природан број k ($k \leq 10^5$). Друга линија стандардног улаза садржи природан број n ($n \leq 10^5$). У следећих n линија налазе се по један природан број (ти бројеви представљају редом елементе низа a).

Излаз: На стандарном излазу приказати колико постоји сегмената низа a чије су суме дељиве са k.

Пример

Улаз	*Излаз*
3 5 1 8 2 3 4	4

Објашњење: то су сегменти $(1, 8)$, (3), $(2, 3, 4)$ и $(1, 8, 2, 3, 4)$

Решење

Груба сила

Директан начин да се задатак реши је да се угнежђеним петљама наброје сви сегменти, да се за сваки израчуна сума и да се провери да ли је дељива са k, бројећи успут такве сегменте. Број је дељив са k ако и само ако

даје остатак 0 при дељењу са k, а знамо да је остатак при дељењу збира са бројем k заправо једнак збиру остатака тј. да важи да је $(a + b) \bmod k = (a \bmod k + b \bmod k) \bmod k$.

Дакле, за сваки сегмент довољно је израчунати збир по модулу k, при чему за фиксирани леви крај сегмента, збир сваког наредног сегмента $[i, j]$ добијамо инкрементално, од збира сегмента $[i, j - 1]$, додајући на њега број a_j и израчунавајући остатак добијеног збира при дељењу са k. Збир префикса смо рачунали инкрементално, на пример, у задатку Префикс највећег збира.

Уз овако инкрементално израчунавање збира, сложеност алгоритма једнака је броју сегмената што је $O(n^2)$.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ucitavamo ulazne podatke
  int k;
  cin >> k;
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // broj segmenata deljivih sa k
  int broj = 0;
  // obradjujemo sve segmente [i, j]
  for (int i = 0; i < n; i++) {
    // zbir po modulu k segmenta [i, j] inicijalizujemo na nulu
    int s = 0;
    for (int j = i; j < n; j++) {
      // azuriramo zbir po modulu k segmenta [i, j] na osnovu zbira [i, j-1]
      s = (s + a[j]) % k;
      // ako je zbir po modulu k jednak 0, zbir je deljiv sa k
      if (s == 0)
        broj++;
    }
  }

  // ispisujemo konacan broj segmenata
  cout << broj << endl;

  return 0;
}
```

Остаци збирова префикса

Задатак можемо решити веома слично задатку Сегмент највећег збира дељивог дужином низа.

Тражимо број сегмената $a_p, a_{p+1}, \ldots, a_q$, за $0 \leq p \leq q < n$, таквих да је збир $S_{pq} = a_p + a_{p+1} + ... + a_q$ дељив са k.

Обележимо са $S_0 = 0$, $S_1 = a_0$, $S_2 = a_0 + a_1$, итд., тј. обележимо са S_i, $0 < i \leq n$ збир првих i елемената низа ($S_i = a_0 + a_1 + ... + a_{i-1}$). Збир S_{pq} можемо изразити као $S_{q+1} - S_p$. Сличну технику користили смо, на пример, у задатку Аритметички троугао или Сегмент датог збира у низу целих бројева.

На основу особина операција по модулу важи да је $S_{pq} \bmod k = (S_{q+1} \bmod k - S_p \bmod k + k) \bmod k$.

Према томе збир S_{pq} је дељив са k (тј. важи $S_{pq} \bmod k = 0$) акко збирови S_{q+1} и S_p имају исти остатак при дељењу са k тј. ако је $S_{q+1} \bmod k = S_p \bmod k$.

Обележимо са b_r број збирова S_i (за $0 \leq i \leq n$) који при дељењу са k дају остатак r (за свако $0 \leq r < k$). Сваки пар (различитих) збирова префикса који дају исти остатак r одређује тачно један сегмент чији је збир елемената дељив са k. У скупу од m различитих елемената постоји тачно $\frac{m(m-1)}{2}$ различитих парова. Зато је за свако r број сегмената који се добија комбинујући два збира који дају остатак r једнак $\frac{b_r \cdot (b_r-1)}{2}$. Према томе укупан број сегмената дељивих са k је $\sum_{r=0}^{k-1} \frac{b_r \cdot (b_r-1)}{2}$.

Остаје још само питање како избројати збирове префикса за сваки дати остатак тј. како израчунати све бројеве b_r. Низом b дужине k памтимо број префикса чији збирови елемената дају остатке редом $0, 1, 2, \ldots, k-1$ тако да је b_r једнак броју префикса чији збир елемената при дељењу са k даје остатак r (што је могуће, с обзиром на дату горњу границу броја k). Збирове префикса, наравно, израчунавамо инкрементално. Слично смо радили, на пример, у задатку Префикс највећег збира.

Полазни збир је $S_0 = 0$, тако да све елементе низа b иницијализујемо на 0, осим вредности на позицији 0 коју иницијализујемо на 1. Збир текућег префикса одржавамо у променљивој S коју иницијализујемо на нулу. Учитавамо члан по члан низа x, и при томе ажурирамо збир префикса (S постављамо на $(S + x) \bmod k$), увећавајући одговарајући бројач (вредност b_S увећавамо за 1).

Прикажимо рад овог алгоритма на примеру одређивања броја сегмената низа $1, 8, 2, 3, 4$ који су дељиви са 3. Могући остаци су 0, 1 и 2.

```
i  ai Si  b0 b1 b2
      0   1  0  0
0  1  1   1  1  0
1  8  0   2  1  0
2  2  2   2  1  1
3  3  2   2  1  2
4  4  0   3  1  2
```

Зато је коначан резултат $\frac{b_0(b_0-1)}{2} + \frac{b_1(b_1-1)}{2} + \frac{b_1(b_1-1)}{2} = 3 + 0 + 1 = 4$.

Временска сложеност овог алгоритма је, јасно, $O(n)$. Приметимо да у овом решењу није било потребно користити низ за бројеве које уносимо, јер при уносу обрадимо сваки елемент, међутим, користимо помоћни низ дужине k, па је меморијска сложеност $O(k)$. Обратимо пажњу на то да ово може бити ограничавајући фактор, ако k може бити јако велики број (што у овом задатку није случај).

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
  int k;
  cin >> k;

  int n;
  cin >> n;

  // na mestu i u nizu br čuvamo broj segmenata čiji zbir pri
  // deljenju sa k daje ostatak i
  vector<int> br(k, 0);
  br[0] = 1;

  // zbir tekućeg segmenata
  int s = 0;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    // ažuriramo zbir elemenata tekućeg segmenta po modulu k
    s = (s + x) % k;
    br[s]++;
  }
```

```
  // izračunavamo ukupan broj segmenata deljivih sa k
  int rez = 0;
  for(int i = 0; i < k; i++)
    rez += br[i]*(br[i]-1)/2;

  cout << rez << endl;

  return 0;
}
```

Задатак: Сегмент највећег збира дељивог дужином низа

Дат је низ природних бројева. Одредити сегмент (подниз узастопних елемената) који има највећи збир који је дељив дужином тог низа.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим n елемената низа (бројеви између 1 и 100, сви у истом реду, раздвојени размацима).

Излаз: На стандардни излаз исписати највећи збир сегмента који је дељив бројем n.

Пример

Улаз	*Излаз*
10 14 8 9 6 6 13 12 7 6 17	50

Задатак: Први сегмент дељив са N

За низ целих позитивних бројева исписати први сегмент (подниз узастопних елемената) чија је сума дељива са n без остатка. Сегмент је први, ако не постоји ни један такав сегмент који се завршава пре њега, нити постоји ни један сегмент који се завршава на истом месту као он, али почиње пре њега.

Улаз: У првој линији стандардног улаза налази се дужина низа представљена природним бројем n ($2 \leq n \leq 10^5$). У следећих n линија налази се по један природан број (ти бројеви представљају редом елементе низа a).

Излаз: У свакој линији стандардног излаза исписује се по један елемент траженог сегмента.

Пример

Улаз	*Излаз*
10 5 7 1 3 5 6 8 9 9 1	7 1 3 5 6 8

Задатак: Слаткиши за сав новац

Дуж једне улице деца продају слаткише. Јована има G динара и жели да их потроши тако што ће кренути да се шета дуж улице и од сваког детета, редом, ће купити тачно један слаткиш (ни једно дете неће прескочити). Ако су познате цене свих слаткиша и ако је позната позиција са које Јована креће у куповину, одредити број слаткиша које ће Јована на тај начин купити.

Улаз: Са стандардног улаза се уноси број деце који продају слаткише n ($1 \leq n \leq 5 \cdot 10^4$), а затим и низ који садржи цене слаткиша (позитивни природни бројеви мањи од 100). Након тога се уноси број упита k ($1 \leq k \leq 5 \cdot 10^4$), а затим у k наредних редова упити који садрже позицију првог детета које ће Јована обићи (позиције се броје од нуле), а затим број динара које Јована има.

Излаз: На стандардни излаз за сваки упит у посебном реду исписати број слаткиша које ће Јована купити.

Пример

Улаз	*Излаз*
7 3 5 1 2 3 1 4 4 3 1 2 5 2 13 0 10	0 2 5 3

Објашњење

У првом упиту Јована има један динар, а креће од детета које продаје слаткиш за 2 динара, па не може да купи ни један слаткиш.

У другом упиту Јована има пет динара и купује слаткише који коштају 1 и 2 динара.

У трећем упиту Јована има 13 динара и купује слаткише који коштају 1, 2, 3, 1 и 4 динара.

У четвртом упиту Јована има 10 динара и купује слаткие који коштају 3, 5 и 1 динар.

Задатак: Кружни пут

На кружном путу налази се n бензинских станица. За сваку станицу познат је број литара бензина који се може на њој купити и растојање до следеће станице. Претпостављамо да је капацитет резервоара аутомобила неограничен и да је за једну јединицу растојања потребан 1 литар бензина. Написати програм којим се одређује редни број прве станице одакле можемо аутомобилом прећи цео круг (ако је то могуће). На почетку обиласка резервоар аутомобила је празан и пуни се на почетној станици.

Улаз: Прва линија стандардног улаза садржи број станица n, ($2 \leq n \leq 10^5$). У наредних n линија налази се за сваку станицу растојање до следеће станице и количина горива која се може купити (цели бројеви између 1 и 1000, развојени размаком).

Излаз: На стандардном излазу приказати у једној линији редни број прве станице одакле се може обићи круг (станице се броје од 0) или -1 ако кружни обилазак није могућ.

Пример

Улаз	*Излаз*
5 3 4 3 2 3 2 4 7 3 2	3

Решење

Груба сила

Тражење кружног пута можемо урадити исцрпном претрагом, тако што од сваке станице редом кренемо обилазак из почетка и видимо до које станице можемо доћи. Ако кренувши из неке станице успемо да се вратимо у почетну станицу (ако успемо да извршимо n прелазака), успешно смо пронашли почетну станицу из које је могуће прећи кружни пут. У сваком кораку проверу да ли можемо ићи даље вршимо тако што проверавамо да ли тренутна количина бензина у резервоару мања или једнака од растојања до следеће станице. Алтернативно, могуће је проверавати да ли је укупан пут од почетне до следеће станице мањи или једнак укупној количини бензина која се може наточити на станицама од почетне до текуће. Прелазак на следећу станицу радимо увећавањем индекса последње станице по модулу n. При преласку у следећу станицу ажурирамо тренутну количину бензина у резервоару тако што одузмемо растојање од претходне станице и додамо бензин који имамо на располагању у станици у коју смо стигли.

Пошто на претходно описан начин више пута пролазимо кроз низ (за сваки почетак највише n пута), сложеност овог приступа је $O(n^2)$.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Stanica {
  int benzin; // kolicina benzina koja se moze natociti
  int rastojanje; // rastojanje do sledece stanice
};

int trazi(vector<Stanica> stanice, int n) {
  // proveravamo svaki moguci pocetak
  for (int poc = 0; poc < n; poc++) {
    int k = poc;  // stanica u kojoj se trenutno nalazimo
    int u_rezervoaru = stanice[k].benzin; // kolicina goriva u rezervoaru
    // pomeramo se dalje dok god imamo goriva i dok ne napravimo n prelaza
    int i;
    for (i = 0; i < n && stanice[k].rastojanje <= u_rezervoaru; i++) {
      // umanjujemo gorivo za prethodni put
      u_rezervoaru -= stanice[k].rastojanje;
      // prelazimo u sledecu stanicu
      k = (k+1) % n;
      // sipamo gorivo u stanici u koju smo dosli
      u_rezervoaru += stanice[k].benzin;
    }
    // ako smo napravili n prelazaka, zavrsili smo kruzni put
    if (i == n)
      return poc;
  }
  return -1;
}


int main() {
  int n;
  cin >> n;
  vector<Stanica> stanice(n);
  for(int i = 0; i < n; i++)
    cin >> stanice[i].rastojanje >> stanice[i].benzin;
  cout << trazi(stanice, n) << endl;
  return 0;
}
```

Збирови префикса и суфикса

Свакој пумпи можемо придружити вредност d_i која представља "допринос" те пумпе осталима односно разлику између бензина купљеног на тој пумпи и растојања до наредне пумпе. Када се пређе са неке пумпе на следећу, количина бензина у резервоару се промени тачно за допринос те пумпе. Дакле, ако саберемо редом доприносе свих пумпи, добијамо количине бензина у резервоару при доласку у сваку наредну пумпу, при чему, да би кружни пут био могућ, та количина никада не сме да буде негативна. Ако обилазак кренемо из пумпе са редним бројем i, тада количина бензина у резервоару редом узима следеће вредности.

$$
\begin{aligned}
&0\\
&d_i\\
&d_i + d_{i+1}\\
&\ldots\\
&d_i + d_{i+1} + \ldots + d_{n-1}
\end{aligned}
$$

$$
\begin{aligned}
&d_i + d_{i+1} + \ldots + d_{n-1} \ + \ d_0\\
&d_i + d_{i+1} + \ldots + d_{n-1} \ + \ d_0 + d_1\\
&\ldots\\
&d_i + d_{i+1} + \ldots + d_{n-1} \ + \ d_0 + d_1 + \ldots + d_{i-1}
\end{aligned}
$$

Све ове вредности ће бити ненегативне ако је најмања од њих ненегативна. Дакле, желимо да будемо у могућности да за произвољну вредност i што брже одредимо најмању од наведених вредности. То је могуће урадити у константном времену, ако урадимо претходно претпроцесирање. Приметимо да смо збирове поделили у две групе. У првој групи се налазе сви парцијални збирови од позиције i надаље, а у другој се налазе збирови добијени сабирањем збира суфикса низа од позиције i и парцијалних збирова од позиције 0 до позиције $i-1$ (збирова префикса до позиције $i-1$). Коришћењем помоћних низова, за сваку позицију i можемо упамтити следеће вредности:

- S_i - сума суфикса који почиње на позицији i,
- S_i^{min} - минимална од свих кумулативних сума које почињу на позицији i,
- P_i^{min} - минимална од свих кумулативних сума префикса закључно са префиксом који се завршава на позицији i.

Све ове низове можемо попунити веома ефикасно, коришћењем принципа инкременталности (као у задацима Префикс највећег збира и Максимални збир сегмента фиксираног почетка).

- Ако је P_i сума префикса до позиције i, важи да је $P_0 = d_0$, да је $P_{i+1} = P_i + d_{i+1}$, за $0 \leq i < n-1$.
- Важи да је $P_0^{min} = P_0$ и да је $P_{i+1}^{min} = \min(P_i^{min}, P_{i+1})$, за $0 \leq i < n-1$.
- Важи и да је $S_{n-1} = d_{n-1}$ и $S_i = d_i + S_{i+1}$, за $i \leq 0 < n-1$.
- Важи и да је $S_{n-1}^{min} = d_{n-1}$ и $S_i^{min} = \min(d_i, d_i + S_{i+1}^{min})$, за $0 \leq i < n-1$.

Са овим вредностима на располагању, лако можемо изразити најмањи збир доприноса пупми, када се пут започне од позиције i: минимум збирова у првој групи је S_i^{min}, а у другој групи је $S_i + P_i^{min}$. Зато за сваку станицу i проверавамо да ли је $S_i^{min} \geq 0$ и да ли је $S_i + P_{i-1}^{min} \geq 0$ (за $i = 0$ довољно је проверити само први услов). Прва станица за коју то важи је она из које можемо направити кружни пут.

Сложеност овог алгоритма је $O(n)$, при чему се у имплементацији користи неколико помоћних низова (но меморијска сложеност остаје $O(n)$).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> dobit(n);
  for (int i = 0; i < n; i++) {
    int rastojanje, benzin;
    cin >> rastojanje >> benzin;
    dobit[i] = benzin - rastojanje;
  }
```

```
  vector<int> minSumaPrefiksa(n);
  minSumaPrefiksa[0] = dobit[0];
  int sumaPrefiksa = dobit[0];
  for (int i = 1; i < n; i++) {
    sumaPrefiksa += dobit[i];
    minSumaPrefiksa[i] = min(minSumaPrefiksa[i-1], sumaPrefiksa);
  }

  vector<int> sumaSufiksa(n);
  vector<int> minSumaSufiksa(n);
  sumaSufiksa[n-1] = dobit[n-1];
  minSumaSufiksa[n-1] = dobit[n-1];
  for (int i = n - 2; i >= 0; i--) {
    sumaSufiksa[i] = dobit[i] + sumaSufiksa[i+1];
    minSumaSufiksa[i] = min(dobit[i], dobit[i] + minSumaSufiksa[i+1]);
  }

  int p = -1;
  for (int i = 0; i < n; i++)
    if (minSumaSufiksa[i] >= 0 &&
        (i == 0 || sumaSufiksa[i] + minSumaPrefiksa[i-1] >= 0)) {
      p = i;
      break;
    }

  cout << p << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Збирови правоугаоника

Сателитски снимак једног правоугаоног подручја одређује плодност тла. Подручје је подељено на одређен број квадрата (површине 10 пута 10 метара) и за сваки квадрат је плодност оцењена бројем од 0 до 10. Напиши програм који на основу таквог снимка може ефикасно да одреди укупну плодност за сваку од неколико датих њива правоугаоног облика.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 1000$) који представља димензију сателитског снимка квадратног облика (изражену бројем квадрата чија је плодност позната), а затим у наредних n редова по n целих бројева између 0 и 10 који представљају оцену плодности на свим појединачним квадратима снимка. Након тога се учитава број m ($1 \leq m \leq 10000$), који представља број њива чија се плодност израчунава, а затим у наредних m линија по четири броја раздвојена са по једним размаком, а који представљају координате два наспрамна темена њиве (изражена бројевима од 0 до $n - 1$). Уноси се прво индекс врсте, па индекс колоне првог темена, а затим индекс врсте, па индекс колоне другог темена.

Излаз: На стандардни излаз исписати m природних бројева (сваки у посебном реду) који представљају оцену плодности сваке њиве. Укупна оцена плодности њиве је збир оцена плодности свих квадрата те њиве.

Пример

Улаз	*Излаз*
5 1 2 3 4 5 5 4 3 2 1 1 2 3 4 5 5 4 3 2 1 1 2 3 4 5 3 0 0 4 4 3 1 1 3 0 3 4 2	75 27 31

Решење

Сателитски снимак представљамо квадратном матрицом $n \times n$ која садржи бројеве од нула до 10.

Груба сила

Директно решење подразумева да након учитавања података теменима њиве, израчунамо збир свих елемената у одговарајућем делу матрице. Ако су учитана темена (i_1, j_1) и (i_2, j_2), тада правоугаонику припадају сва поља са координатама (i, j) за $\min(i_1, i_2) \leq i \leq \max(i_1, i_2)$ и $\min(j_1, j_2) \leq j \leq \max(j_1, j_2)$. Један начин имплементације је да се границе у петљама одређују библиотечким функцијама за израчунавање минимума тј. максимума два броја, а други је да се бројеви уреде по величини тако да је $i_1 \leq i_2$ и $j_1 \leq j_2$.

Учитавање матрице захтева $O(n^2)$ операција, а израчунавање и штампање збирова захтева $O(m \cdot n^2)$ операција.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<vector<int>> A(n);

  for (int i = 0; i < n; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }

  int m;
  cin >> m;
  for (int k = 0; k < m; k++) {
    int zbir = 0;
    int i1, j1, i2, j2;
    cin >> i1 >> j1 >> i2 >> j2;
    for (int i = min(i1, i2); i <= max(i1, i2); i++)
      for (int j = min(j1, j2); j <= max(j1, j2); j++)
        zbir += A[i][j];
    cout << zbir << '\n';
  }

  return 0;
}
```

Директни приступ је могуће значајно оптимизовати, примењујући сличну технику израчунавања збирова одређених "префикса" матрице, слично као у задацима Аритметички троугао или Збирови сегмената. Дефинишимо матрицу Z_{ij} за све вредности $0 \leq i \leq n$ и $0 \leq j \leq n$ на следећи начин. Ако је i или j једнако 0, тада је и $Z_{ij} = 0$. У супротном, Z_{ij} је једнак збиру свих елемената матрице A из појаса ограниченог тачкама са координатама $(0, 0)$ и $(j-1, i-1)$.

Вредности матрице Z_{ij} можемо израчунати инкрементално (слично као у задатку Префикс највећег збира). Након попуњавања граничних вредности нулама, попуњавамо врсту по врсту тако што приметимо да важи да је $Z_{ij} = Z_{(i-1)j} + Z_{i(j-1)} - Z_{(i-1)(j-1)} + A(i-1)(j-1)$. Наиме, збир $Z_{(i-1)j}$ садржи у себи збир $Z_{(i-1)(j-1)}$ и додатно све елементе $A_{i'(j-1)}$ за $0 \leq i' < i-1$. Слично, збир $Z_{i(j-1)}$ садржи у себи збир $Z_{(i-1)(j-1)}$ и додатно све елементе $A_{(i-1)j'}$ за $0 \leq j' < j-1$. Дакле, у збир $Z_{(i-1)j} + Z_{i(j-1)}$ два пута је укључен збир $Z_{(i-1)(j-1)}$ и једном сви елементи последње врсте и колоне правоугаоника чија су темена $(0, 0)$ и $(i-1, j-1)$ осим елемента $A_{(i-1)(j-1)}$. Одатле тврђење једноставно следи.

Када су префикси израчунати, тада се збир елемената у правоугаонику чије је горње лево теме (i_1, j_1), а доње десно теме (i_2, j_2) може израчунати као $Z_{(i_2+1)(j_2+1)} - Z_{(i_2+1)j_1} - Z_{i_1(j_2+1)} + Z_{i_1 j_1}$. Наиме, правоугаоник P са теменима $(0, 0)$ и (i_2, j_2) се може разложити на правоугаоник P_1 са теменима $(0, 0)$ и $(i_1 - 1, j_1 - 1)$, праваоугаоник P_2 са теменима $(i_1, 0)$ и $(i_2, j_1 - 1)$, правоугаоник P_3 са теменима $(0, _1)$ и $(i_1 - 1, j_2)$ и правоугаоник P_4 са теменима (i_1, j_1) и (i_2, j_2). Број $Z_{(i_2+1)(j_2+1)}$ је збир свих елемената у правоугаонику P, број $Z_{(i_2+1)j_1}$ је збир свих елемената у унији правоугаоника P_1 и P_2, број $Z_{i_1(j_2+1)}$ је збир свих елемената у унији правоугаоника P_1 и P_3, док је број $Z_{i_1 j_1}$ једнак збиру свих елемената у правоугаонику P_1. Дакле, израз $Z_{(i_2+1)(j_2+1)} - Z_{(i_2+1)j_1} - Z_{i_1(j_2+1)} + Z_{i_1 j_1}$ је једнак разлици збира свих елемената у правогаонику P и збира свих елемената у правоугаоницима P_1, P_2 и P_3, а то је управо број свих елемената у правоугаонику P_4 (на наредној слици су бројевима обележени елементи правоугаоника P_1, P_2, P_3 и P_4).

```
    i1    i2
1 1 2 2 2 2 . .
1 1 2 2 2 2 . .
3 3 4 4 4 4 . . j1
3 3 4 4 4 4 . .
3 3 4 4 4 4 . . j2
. . . . . . . .
```

Након учитавања матрице за шта је потребно $O(n^2)$ операција, матрица збирова "префикса" се такође израчунава у $O(n^2)$ операција. Након тога, збир сваког правоугаоника се израчунава у константном времену, тако да је додатно време још само $O(m)$ и укупна сложеност оваквог алгоритма је $O(n^2 + m)$.

Ово је задатак у ком се одговара на упите, па је у језику C++ потребно оптимизовати синхронизацију улаза и излаза и прилагодити је аутоматском тестирању, навођењем `cin.tie(0)` и коришћењем `\n` уместо `endl`.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<vector<int>> A(n);

  for (int i = 0; i < n; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }

  vector<vector<int>> Z(n+1);
  Z[0].resize(n+1, 0);
```

```
  for (int i = 1; i <= n; i++) {
    Z[i].resize(n+1);
    Z[i][0] = 0;
    for (int j = 1; j <= n; j++)
      Z[i][j] = Z[i-1][j] + Z[i][j-1] - Z[i-1][j-1] + A[i-1][j-1];
  }

  int m;
  cin >> m;
  for (int k = 0; k < m; k++) {
    int i1, j1, i2, j2;
    cin >> i1 >> j1 >> i2 >> j2;
    if (i1 > i2) swap(i1, i2);
    if (j1 > j2) swap(j1, j2);
    cout << Z[i2+1][j2+1] - Z[i2+1][j1] - Z[i1][j2+1] + Z[i1][j1] << '\n';
  }

  return 0;
}
```

Задатак: Пар производа у ранцу

Ненад је отпутовао у Париз и жели да купи поклоне својим родитељима. Пошто путује нискотарифном авиокомпанијом, ограничена је маса коју може да понесе у свом ранцу. Оцу ће купити поклон у једној, а мајци у другој продавници. Напиши програм који на основу познатих маса и цена поклона у обе продавнице помаже Ненаду да родитељима купи што вредније поклоне (гледајући њихову укупну цену).

Улаз: Са стандардног улаза се уносе подаци у производима у две продавнице. За сваку продавницу се уноси број производа n ($1 \leq n \leq 10^5$), и затим у наредних n линија по два цела броја, сваки између 1 и 10^9, који представљају масу и затим цену производа. Након тога се уноси и укупна маса поклона коју је могуће понети у ранцу (цео број између 1 и $2 \cdot 10^9$).

Излаз: На стандардни излаз исписати највећи могући збир цена првог и другог поклона које је могуће понети у ранцу.

Пример

Улаз	*Излаз*
3 2 4 3 6 4 5 3 2 3 3 2 4 5 6	9

Решење

У овом задатку ефикасно решење захтева комбиновање неколико алгоритамских технника. Иако се на први поглед може помислити да се овај задатак решава као класичан проблем ранца, динамичким програмирањем, велика димензија капацитета ранца указује на то да се задатак мора решавати другачије.

Груба сила

Можемо да проверимо сваки пар предмета (у коме је један предмет из једне, а други из друге продавнице).

Ово решење је квадратне сложености (тј. сложености $O(mn)$ где је m број предмета у првој, а n број предмета у другој продавници), и самим тим је прилично неефикасно.

```
#include <iostream>
#include <vector>
#include <utility>
```

```
using namespace std;

// proizvode predstavljamo uredjenim parom (tezina, cena)
typedef pair<int, int> Proizvod;

// ucitavanje niza proizvoda sa standardnog ulaza
vector<Proizvod> ucitajProizvode() {
  int n;
  cin >> n;
  vector<pair<int, int>> proizvodi(n);
  for (int i = 0; i < n; i++) {
    int cena, tezina;
    cin >> cena >> tezina;
    proizvodi[i] = make_pair(cena, tezina);
  }
  return proizvodi;
}

int main() {
  // ucitavamo podatke o proizvodima i maksimalnu tezinu koja moze da
  // stane u ranac
  vector<Proizvod> proizvodi1 = ucitajProizvode();
  vector<Proizvod> proizvodi2 = ucitajProizvode();
  int maksTezina;
  cin >> maksTezina;

  // ispituju se svi parovi proizvoda i odredjuje maksimalna cena
  // medju onim parovima koji staju u ranac
  int maksCena = 0;
  for (const auto& p1 : proizvodi1) {
    if (p1.first >= maksTezina) continue;
    for (const auto& p2 : proizvodi2)
      if (p1.first + p2.first <= maksTezina &&
          p1.second + p2.second > maksCena)
        maksCena = p1.second + p2.second;
  }
  cout << maksCena << endl;
}
```

Сортирање, бинарна претрага и максимуми префикса

Један начин да се дође до ефикаснијег решења је да подаци буду на неки начин сортирани. Питање је: како сортирати?

Ако бисмо један низ сортирали по маси, за сваки предмет из другог низа бисмо могли пронаћи све предмете који са њим могу стати у ранац. Ипак, ово није довољно да се задатак реши ефикасно, јер масе и цене не расту заједно, па бисмо морали анализирати све те предмете да нађемо најскупљи међу њима.

Уређивање по цени није боље, јер бисмо тада за фиксиран предмет из једне продавнице морали међу онима из друге (који у збиру са првим) поправљају укупну вредност, да тражимо оне који се уклапају по маси, што би опет било веома неефикасно.

Оно што би нам овде помогло (након низа предмета по маси) је да можемо за сваки предмет P из сортираног низа да брзо одговоримо колико вреди највреднији предмет тог низа, чија маса није већа од масе предмета P. Такве одговоре можемо да припремимо након сортирања а пре испитивања парова, тако што формирамо још један низ у коме ће се ти одговори наћи. Конкретније, можемо у једном пролазу кроз низ сортиран по маси да за сваки префикс тог низа израчунамо цену највреднијег предмета у префиксу. Префикс рачунамо инкрементално, слично као у задатку Најбољи "сабмит". Нека је m_i максимум префикса $[0, i)$. Тада је $m_0 = 0$ (пошто су сви елементи позитивни, вредност 0 може играти улогу $-\infty$, што је уобичајена иницијализација максимума празног низа). Важи да је $m_{i+1} = \max(m_i, a_i)$. Приметимо одређену сличност са задатком Збирови сегмената где смо претпроцесирањем низа формирали збирове префикса, који су нам касније по-

могли да брзо одговоримо на упите. На сличан начин нам израчунавање максимума префикса током фазе претпроцесирања помаже да касније брзо одговоримо на упите.

Довољно је да сортирамо само други низ и израчунамо максимуме његових префикса, као у претходном решењу. Након тога можемо за сваки предмет неуређеног првог низа да бинарном претрагом нађемо предмет највеће масе из другог низа, који се може понети заједно са првим, а онда (користећи максимуме префикса другог низа) да нађемо и вредност највреднијег предмета у другом низу, који се може понети заједно са текућим предметом из првог низа.

Сложеност сортирања је $O(n \log n)$, након чега се врши n бинарних претрага, при чему је сака сложености $O(\log n)$. Дакле, и прва и друга фаза се извршавају у сложености $O(n \log n)$, што је и укупна сложеност овог решења.

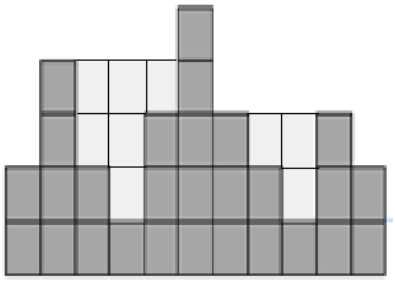
```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>

using namespace std;

// proizvode predstavljamo uredjenim parom (tezina, cena)
typedef pair<int, int> Proizvod;

// ucitavanje niza proizvoda sa standardnog ulaza
vector<Proizvod> ucitajProizvode() {
  int n;
  cin >> n;
  vector<Proizvod> proizvodi(n);
  for (int i = 0; i < n; i++) {
    int tezina, cena;
    cin >> tezina >> cena;
    proizvodi[i] = make_pair(tezina, cena);
  }
  return proizvodi;
}

int main() {
  // ucitavamo podatke o proizvodima i maksimalnu tezinu koja moze da
  // stane u ranac
  vector<Proizvod> proizvodi1 = ucitajProizvode();
  vector<Proizvod> proizvodi2 = ucitajProizvode();
  int maksTezina;
  cin >> maksTezina;

  // sortiramo proizvode iz druge prodavnice po tezini
  sort(begin(proizvodi2), end(proizvodi2));

  // za svaku poziciju u sortiranom nizu proizvoda iz druge prodavnice
  // odredjujemo maksimalnu cenu proizvoda striktno pre te pozicije
  vector<int> maksCenaDo(proizvodi2.size() + 1);
  maksCenaDo[0] = 0;
  for (size_t i = 1; i <= proizvodi2.size(); i++)
    maksCenaDo[i] = max(maksCenaDo[i-1], proizvodi2[i-1].second);

  // maksimalni zbir cena dva proizvoda koja mogu da stanu u ranac
  int maksCena = 0;
  // analiziramo jedan po jedan proizvod iz prve prodavnice
  for (const auto& p1 : proizvodi1) {
    // odredjujemo najvecu poziciju pre koje se svi proizvodi iz druge
    // pozicije mogu staviti u ranac sa tekucim proizvodom iz prve
```

```
    // prodavnice
    auto p2Granica = upper_bound(begin(proizvodi2), end(proizvodi2),
                                 maksTezina - p1.first,
                                 [](int tezina, const Proizvod& pj) {
                                     return tezina < pj.first;
                                 });
    int pj = distance(begin(proizvodi2), p2Granica);
    // ako ima bar neki takav proizvod
    if (pj > 0)
      // kombinujemo tekuci proizvod iz prve prodavnice sa najskupljim
      // proizvodom iz druge pre te pozicije (on se sigurno moze
      // iskombinovati sa tekucim proizvodom iz prve prodavnice)
      maksCena = max(maksCena, p1.second + maksCenaDo[pj]);
  }

  // ispisujemo pronadjeni zbir
  cout << maksCena << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Попуњавање празнина

У магацину се налази неколико постоља која су поређана једно поред другог. На сваком постољу се, једна на другој, налазе кутије истих димензија чији број је за свако постоље задат одговарајућим чланом низа. Одредити колико кутија највише можемо додати тако да се попуне празнине настале слагањем кутија (кутија која се дода у празнину се налазе између већ постојећих кутија и не виде се ни са лева, ни са десна).



Слика 2.13: Постоља и кутије

Улаз: У првој линији стандардног улаза налази се број постоља, N ($3 \leq N \leq 50000$), а затим, у свакој од наредних N линија стандардног улаза, број кутија на сваком постољу (природан број између 0 и 100).

Излаз: У једној линији стандарног излаза налази се цео број већи или једнак 0 који представља највећи број кутија које можемо додати.

Пример1

Улаз	*Излаз*
11 2 4 2 1 3 5 3 2 1 3 2	9

Пример2

Улаз	*Излаз*
11 2 4 2 5 3 2 3 5 1 3 2	11

Задатак: Збир подниза карата

Ђурица је пронашао n карата поређаних у низ. На картама су записани цели бројеви. Датом низу карата A, Ђурица доделује вредност $f(A) = A_{n-1} - A_0$, која је једнака разлици вредности прве и последње карте у низу.

Ђурица из низа жели да избаци највише M карата. Избацивањем неких m карата, $m \leq M$, добија нов низ карата којем поново рачуна вредност на описани начин. Од свих могућих одабира вредности m и свих могућих одабира m карата које ће избацити, њега занима онај низ карата који ће имати највећу вредност. Помозите Ђурици и реците му колико је та највећа вредност. Приметимо да Ђурица ни у једном моменту не мења распоред карата датих на почетку.

Улаз: Прва линија стандардног улаза садржи број n ($2 \leq n \leq 10^5$), а наредни n бројева на картама, раздвојених са по једним размаком (бројеви су између -10^6 и 10^6). У трећој линији налази се број M ($0 \leq M \leq n-2$).

Излаз: На стандардни излаз исписати највећу вредност функције f, када се из низа избаци између 0 и M карата.

Пример 1

Улаз	*Излаз*
11 5 1 7 2 5 3 8 2 3 6 5 4	5

Објашњење

Највећа вредност функције се може добити, на пример, када се избаце три карте са вредношћу 5.

Пример 2

Улаз

```
10
10 9 8 7 6 5 4 3 2 1
3
```

Излаз

```
-6
```

Задатак: Највећи НЗД низа након замене једног елемента

Напиши програм који одређује највећи могући највећи заједнички делилац елемената низа ако је у низу допуштена замена тачно једног елемента произвољном вредношћу.

Улаз: Са стандардног улаза се учитава број n ($3 \leq n \leq 10^5$), а затим у наредном реду n природних бројева мањих од 10^{15}.

Излаз: На стандрдни излаз исписати тражени НЗД.

Пример 1

Улаз	*Излаз*
3 6 7 8	2

Објашњење

Ако број 7 заменимо бројем 2, добијамо низ 6, 2, 8, чији је НЗД једнак 2. Није могуће добити НЗД већи од 2.

Пример 2

Улаз

```
3
12 18 17
```

Излаз

```
6
```

Објашњење

Елемент 17 можемо, на пример, заменити бројем 30 и тада је НЗД низа 12, 18, 30 једнак 6. Није могуће добити НЗД већи од 6.

Решење

Груба сила

Основни увид за решење задатка је тај да се заменом неког елемента низа не може добити НЗД који је већи од НЗД-а преосталих елемената низа. Са друге стране тај НЗД преосталих се увек може достићи (ако се баш он упише на место замењеног елемента). Дакле, у задатку је потребно избацивати један по један елемент низа и одређивати НЗД преосталих елемената и међу тим НЗД-овима одредити највећи.

Решење грубом силом анализира један по један елемент и сваки пут изнова рачуна НЗД свих елемената осим тог.

Ако претпоставимо да је проналажење НЗД два броја операција константне сложености, сложеност одређивања НЗД преосталих $n-1$ елемената низа врши се у времену $O(n)$. Пошто се ово ради за сваки елемент низа, сложеност је $O(n^2)$. Меморијска сложеност је $O(n)$ и складишти се само оригинални низ.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// izracunavanje nzd Euklidovim algoritmom
long long nzd(long long a, long long b) {
  while (b != 0) {
    long long tmp = a % b;
    a = b;
    b = tmp;
  }
  return a;
}

int main() {
  // ucitavamo niz brojeva
  int n;
  cin >> n;
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
```

```
  // izracunavamo maksimalni nzd nakon izbacivanja jednog elementa
  long long maks_nzd = 0;
  for (int i = 0; i < n; i++) {
    // nzd inicijalizujemo na bilo koji element koji nije izbacen
    long long nzd_i = i == 0 ? a[1] : a[0];
    // trazimo nzd sa svim ostalim neizbacenim elementima
    for (int j = 0; j < n; j++)
      if (j != i)
        nzd_i = nzd(nzd_i, a[j]);
    // azuriramo maksimum
    maks_nzd = max(maks_nzd, nzd_i);
  }

  cout << maks_nzd << endl;

  return 0;
}
```

Инкременталност

НЗД свих елемената без избаченог можемо израчунати тако што пронађемо посебно НЗД свих елемената испред избаченог, посебно НЗД свих елемената иза избаченог и затим пронађемо НЗД та два броја (изузетак су случајеви када се избацује први тј. последњи елемент низа и онда је довољно наћи само један од ова два броја). Зато за сваки елемент низа треба да знамо НЗД префикса пре тог елемента и НЗД суфикса иза тог елемента. НЗД свих префикса и НЗД свих суфикса можемо рачунати инкрементално. Наиме, важи да је $P_0 = a_0$ и $P_{i+1} = nzd(P_i, a_{i+1})$ и да је $S_{n-1} = a_{n-1}$ и $S_{i-1} = nzd(a_{i-1}, S_i)$, где смо са P_i означили $nzd(a_0, \ldots, a_i)$, а са S_i означили $nzd(a_i, \ldots, a_{n-1})$.

У првој фази инкрементално израчунавамо НЗД свих префикса и свих суфикса и памтимо их у помоћним низовима. У другој фази редом избацујемо један по један елемент и рачунамо НЗД свих преосталих елемената, консултујући низове НЗД-ова префикса и суфикса.

Пошто се НЗД-ови префикса и суфикса рачунају инкрементално, они се сви могу израчунати у времену $O(n)$. За сваки елемент се НЗД преосталих елемената сада може израчунати у времену $O(1)$, очитавајући НЗД префикса и суфикса и израчунавањем њиховог НЗД. Анализира се n елемената, па се и друга фаза извршава у времену $O(n)$. Укупна сложеност је зато $O(n)$. Поред оригиналног, складиштимо још низове суфикса и префикса, али меморијска сложеност остаје $O(n)$. Један од тих низова би било могуће изоставити (на пример, суфиксе је могуће обрадити унапред, а префиксе током изостављања једног по једног елемента).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// izracunavanje nzd Euklidovim algoritmom
long long nzd(long long a, long long b) {
  while (b != 0) {
    long long tmp = a % b;
    a = b;
    b = tmp;
  }
  return a;
}

int main() {
  // ucitavamo niz brojeva
  int n;
  cin >> n;
```

```
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // izracunavamo nzd-ove svih prefiksa niza a
  vector<long long> nzd_prefiks(n);
  nzd_prefiks[0] = a[0];
  for (int i = 1; i < n; i++)
    nzd_prefiks[i] = nzd(nzd_prefiks[i-1], a[i]);
  // izracunavamo nzd-ove svih sufiksa niza a
  vector<long long> nzd_sufiks(n);
  nzd_sufiks[n-1] = a[n-1];
  for (int i = n-2; i >= 0; i--)
    nzd_sufiks[i] = nzd(a[i], nzd_sufiks[i+1]);

  // izracunavamo maksimalni nzd nakon izbacivanja jednog elementa

  // ako se izbace prvi ili poslednji element niza
  long long maks_nzd = max(nzd_prefiks[n-2], nzd_sufiks[1]);
  // ako se izbaci bilo koji element iz sredine niza
  for (int i = 1; i < n-1; i++)
    maks_nzd = max(maks_nzd, nzd(nzd_prefiks[i-1], nzd_sufiks[i+1]));

  cout << maks_nzd << endl;

  return 0;
}
```

Задатак: Увећавање сегмената

Камион превози терет током N километара пута. На пут креће празан и током пута утоварује и истоварује пакете. Ако се за сваки пакет зна на ком је километру пута утоварен, на ком је километру пута истоварен и колика му је маса, напиши програм који одређује колико је оптерећење камиона на сваком километру пута. Сматрати да се предмет утоварује на почетку, а истоварује на крају датог километра.

Улаз: Са стандардног улаза се уноси број километара N ($10 \leq N10000$), затим, у наредном реду, број предмета M ($0 \leq M \leq 10000$), а након тога, у наредних M редова по три цела броја раздвојена размацима који представљају број километра на чијем је почетку утоварен предмет (цео број између 0 и $N-1$), број километра на чијем крају је истоварен (цео број између 0 и $N-1$) и на крају маса предмета (цео број између 1 и 10).

Излаз: На стандардни излаз исписати масу терета у килограмима на сваком километру пута (иза сваке масе написати по један размак).

Пример

Улаз
```
10
3
1 5 10
3 7 10
2 8 15
```

Излаз
```
0 10 25 35 35 35 25 25 15 0
```

Објашњење

```
        km   0   1  2  3  4  5  9  7  8  9
             0   0  0  0  0  0  0  0  0  0
1 5 10       0  10 10 10 10 10  0  0  0  0
3 7 10       0  10 10 20 20 20 10 10  0  0
2 8 15       0  10 25 35 35 35 25 25 15  0
```

Решење

Овај задатак је веома сличан задатку Најбројнији пресек интервала, па се могу користити исте технике решавања.

Директно решење

Директан начин је да се одржава низ M у којем се памти маса на камиону током сваког километра пута. Након учитавања сваког податка о предмету (почетног километра a, завршног километра b и масе m), све вредности у низу M на позицијама од a до b (укључујући и њих) се увећавају за m.

Проблем са овим решењем је то што предмети могу путовати велики број километара па се у сваком кораку врши ажурирање великог броја чланова низа (сложеност је у најгорем случају $O(n \cdot m)$).

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> mase(n, 0);
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    int km_od, km_do, masa;
    cin >> km_od >> km_do >> masa;
    for (int km = km_od; km <= km_do; km++)
      mase[km] += masa;
  }

  for (int masa : mase)
    cout << masa << " ";

  return 0;
}
```

Разлике суседних елемената низа

Задатак можемо решити ефикасније ако уместо да у низу M одржавамо масу у камиону у километру i, одржавамо разлику између масе у километру i и $i-1$ (на позицији 0 се чува маса у камиону у нултом километру). Дакле, уводимо низ R_i такав да је $R_0 = M_0$, а $R_i = M_i - M_{i-1}$, за $1 \leq i < n$. Посматрајмо шта се дешава са низом R када се у низу M сви елементи на позицијама a до b увећају за m.

- Вредност R_a једнака је разлици $M_a - M_{a-1}$ (или евентуално M_0 ако је $a = 0$) и она се увећава за m, јер је M_a увећан за m, док се M_{a-1} не мења.
- Све вредности од R_{a+1} до R_b остају не промењене. Наиме, за све њих важи да је $R_i = M_i - M_{i-1}$, а да су и M_i и M_{i-1} увећани за m.
- На крају, вредност R_{b+1} се умањује за m. Наиме важи да је $R_{b+1} = M_{b+1} - M_b$, да се M_b увећава за m, док се M_{b+1} не мења.

Рецимо да ако је $b = n - 1$, тада не морамо разматрати вредност $R_{b+1} = R_n$ (мада, униформности ради, можемо, што захтева да низ R садржи $n + 1$ елемент). Дакле, приликом сваког учитавања бројева a, b и m потребно је само да увећавамо елемент R_a за m, а да елемент R_{b+1} умањимо за m.

Када знамо елементе низа R елементе низа M можемо једноставно реконструисати сабирањем елемената низа R. Наиме, важи да је $M_0 = R_0$, док је $M_i = M_{i-1} + R_i$, тако да сваки наредни елемент низа M можемо израчунати као збир претходног елемента низа M и њему одговарајућег елемента низа R. Приметимо да је заправо елемент M_i једнак збиру свих елемената од R_0 до R_i, јер је $R_0 + R_1 + \ldots + R_i = M_0 + (M_1 - M_0) + \ldots + (M_i - M_{i-1}) = M_i$.

Укупна сложеност овог приступа је $O(n + m)$.

Идеја коришћена у овом задатку донекле је слична (заправо инверзна) техници одређивања збира сегмената као разлике два збира префикса. Ту технику смо, на пример, применили у задатку Збирови сегмената. Може се приметити да се реконструкција низа врши заправо израчунавањем префиксних збирова низа разлика суседних елемената, што указује на дубоку везу између ове две технике. Заправо, разлике суседних елемената представљају одређени дискретни аналогон извода функције, док префиксни збирови представљају аналогију одређеног интеграла. Израчунавање збира сегмента као разлике два збира префикса одговара Њутн-Лајбницовој формули.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> razlika(n+1, 0);
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    int km_od, km_do, masa;
    cin >> km_od >> km_do >> masa;
    razlika[km_od] += masa;
    razlika[km_do+1] -= masa;
  }

  int masa_km = 0;
  for (int km = 0; km < n; km++) {
    masa_km += razlika[km];
    cout << masa_km << " ";
  }

  return 0;
}
```

Задатак: Најбројнији пресек интервала

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Пермутација са највећим збиром упита

Дуж улице се постављају светиљке које емитују различиту количину осветљења. Светиљке осветљавају и станове који се налазе у близини. За сваки стан је познато које га светиљке осветљавају (то су увек неке узастопне светиљке у низу светиљки распоређених дуж улице). Укупна количина осветљења које стан прима једнака је збиру количине осветљења светиљки које га осветљавају. Ако су познате количине осветљења које емитују све светиљке које треба распоредити, написати програм који одређује распоред светиљки тако да станови укупно буду осветљени што је више могуће.

Улаз: Са стандардног улаза се уноси број светиљки n ($1 \leq n \leq 10^5$), а затим у наредном реду и јачине светиљки које се распоређују (цели бројеви између 1 и 10^6). Након тога се уноси број станова s ($1 \leq s \leq 10^5$) и затим у наредних s редова, за сваки стан информација о позицијама прве и последње светиљке која га осветљава (два броја $[l, d]$, где је $1 \leq l \leq d \leq n$).

Излаз: На стандардни излаз исписати највећу могућу укупну количину светлости коју могу добити сви станови.

Пример 1

Улаз	*Излаз*
3 5 3 2 3 1 2 2 3 1 3	25

Пример 2

Улаз	*Излаз*
5 5 2 4 1 3 3 1 5 2 3 2 3	33

2.13 Техника два показивача

Угнежђене петље обично подразумевају постојање две бројачке променљиве од којих спољашња само увећава своју вредност током итерације, док се вредност бројачке петље у унутрашњој увећава до неке горње границе, затим се поново враћа на неку доњу границу и поново увећава и то се понавља више пута, све док спољашња бројачка променљива не достигне своју максималну вредност. Ово по правилу доводи до квадратне сложености (тј. сложености вишег степена у случају угнежђавања већег броја петљи).

Техника два показивача обухвата широку класу ефикасних алгоритама које такође карактерише постојање две или више бројачких променљивих, које се крећу кроз елементе неког низа (често сортираног). Међутим, оно што је карактеристично за њих је то што се, за разлику од унутрашњих променљивих у угнежђеним петљама, ове променљиве стално "крећу у истом смеру", тј. вредност им се или стално повећава или стално смањује (а честа је и комбинација где се "низ обилази са два краја", где се једна променљива стално повећава, а друга стално смањује). Техничка реализација може бити било помоћу једне петље која контролише вредности обе променљиве, било помоћу угнежђених петљи, али тако да се након завршетка тела унутрашње петље, спољашња променљива увећава до места где се унутрашња петља завршила. Пошто се свака променљива може променити највише n пута (где је n неко горње ограничење њихове вредности, обично дужина низа), број промена (па самим тим и извршавања тела петље) је највише $2n$ и линеаран је по n тј. сложеност му је $O(n)$.

Алгоритми засновани на техници два показивача обично могу да се изведу коришћењем одсецања примењених на угнежђене петље, па је, као и код сваке примене одсецања, потребно пажљиво образложити њихову коректност.

Задатак: Обједињавање

У школи малих жутих мрава наставник је прегледао контролни задатак. Прво је прегледао ђаке који су радили групу А, а затим оне који су радили групу Б, средио је резултате за сваку групу и мраве поређао на основу броја поена који су освојили. Напиши програм који му помаже да од уређеног списка ученика који су радили задатке из групе А и од уређеног списка ученика који су радили задатке из групе Б добије јединствен уређен списак свих ученика.

Улаз: Са стандардног улаза се уноси број ђака m који су радили групу А ($5 \leq m \leq 25000$), а затим неопадајуће сортиран низ поена тих ђака (елементи су у једној линији, раздвојени са по једним размаком). Након тога се уноси број n ђака који су радили групу Б ($5 \leq n \leq 25000$), а затим неопадајуће сортиран низ поена тих ђака (елементи су у једној линији, раздвојени са по једним размаком).

Излаз: На стандардни излаз исписати неопадајуће сортирани низ поена свих ђака заједно, раздвојене са по једним размаком.

Пример

Улаз	*Излаз*
4 1 3 5 7 3 2 4 5	1 2 3 4 5 5 7

Решење

Сортирање

Наиван начин да се задатак реши је да се елементи оба учитана низа прекопирају у трећи (било помоћу петље, било библиотечком функцијом `copy`) и да се онда сортирају, најбоље библиотечком функцијом. Разни начини сортирања описани су у задатку Сортирање бројева.

`sort`).

Копирање низова дужине m и n захтева $m + n$ операција, а сортирање $O((m + n) \cdot \log(m + n))$. Технички, могли смо одмах елементе учитавати у резултујући низ и тако уштедети меморију и време потребно за копирање, али доминатни фактор, а то је време потребно за сортирање би остао. Приметимо да у овом решењу нисмо уопште употребили чињеницу да су полазни елементи већ сортирани.

Иако ово решење по времену извршавања не заостаје пуно у односу на оптимално (његово време извршавања је квазилинеарно тј. $O((m + n) \cdot \log(m + n))$, а оптимално време је линеарно тј. $O(m + n)$), оно је доста компликованије него што је потребно. То се у овој имплементацији не види, јер је употребљена библиотечка функција сортирања, међутим, имплементација ефикасног алгоритма сортирања захтева напредне технике програмирања. Интересантно, један од популарних алгоритама сортирања је сортирање обједињавањем (енгл. merge sort) који као свој основни корак захтева извршавање обједињавања два сортирана низа у трећи. Самим тим, донекле је бесмислено проблем обједињавања решавати свођењем на компликованији проблем сортирања.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo prvi niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ucitavamo drugi niz
  int m;
  cin >> m;
  vector<int> b(m);
  for (int i = 0; i < m; i++)
    cin >> b[i];

  // kopiramo dva niza u treci, jedan iza drugog
  vector<int> c(m + n);
  copy(a.begin(), a.end(), c.begin());
  copy(b.begin(), b.end(), next(c.begin(), n));
  // sortiramo treci niz
  sort(c.begin(), c.end());

  // ispisujemo rezultat
  for (int i = 0; i < c.size(); i++)
    cout << c[i] << " ";
  cout << endl;

  return 0;
}
```

Алгоритам обједињавања

Задатак можемо решити ефикасним алгоритмом, заснованом на техници два показивача. *Алгоритам обједињавања* (енгл. *merge*) подразумева да су низови који се обједињавају сортирани. Ако је један од низова празан, резултат обједињавања је други низ и његове елементе је потребно једноставно прекопирати у резултат. Ако низови нису празни, пошто су сортирани, први елемент низа је уједно најмањи у њему. Мањи од два почетна елемента је мањи (или једнак) од почетног елемента другог низа, па је мањи или једнак свим

елементима у оба низа и самим тим је најмањи елемент од свих и треба да буде први у резултату. Када се тај елемент уклони из низа, добијамо проблем истог типа као и полазни, који се онда решава на исти начин. Имплементација може бити рекурзивна, међутим, рекурзија је репна и лако се елиминише.

Током итеративне имплементације одржавају се два показивача: променљива i која указује на позицију текућег елемента првог и j која указује на текући елемент другог низа. Док су обе ове променљиве мање од дужине низа по којем се крећу, поредимо елементе на тим позицијама. Ако је елемент на позицији i у првом низу мањи (или једнак) елементу на позицији j у другом низу, тада тај елемент преписујемо у трећи низ (на позицију k коју иницијализујемо на нулу и увећавамо приликом додавања сваког новог елемента) и увећавамо i за 1. У супротном у трећи низ преписујемо елемент из другог низа са позиције j и увећавамо j. Када бар једна од променљивих достигне дужину одговарајућег низа, тада елементе преосталог низа преписујемо у трећи низ. Не морамо експлицитно проверавати да ли у неком од ових низова има преосталих елемената, већ можемо у једној петљи копирати преостале елементе првог, а у другој петљи копирати преостале елементе другог низа (једна од ових петљи ће бити празна).

Прикажимо рад овог алгоритма и на једном примеру.

- Претпоставимо да је потребно објединити наредна два низа.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   _ _ _ _ _ _ _ _ _
i           j         k
```

- У првом кораку је $i = 0$ и $j = 0$, па се пореде елементи на позицијама 0, тј. елементи 1 и 2. Пошто је 1 мањи, он се преписује у резултујући низ и увећава се леви показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 _ _ _ _ _ _ _ _
  i         j           k
```

- Сада је $i = 1$ и $j = 0$, па се пореде елементи на позицијама 1 и 0, тј. 3 и 2. Пошто је 2 мањи, он се преписује у резултујући низ и увећава се десни показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 _ _ _ _ _ _ _
  i           j           k
```

- Сада је $i = 1$ и $j = 1$, па се пореде елементи на позицијама 1 и 1, тј. 3 и 4. Пошто је 3 мањи, он се преписује у резултујући низ и увећава се леви показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 _ _ _ _ _ _
    i         j             k
```

- Сада је $i = 2$ и $j = 1$, па се пореде елементи на позицијама 2 и 1, тј. 6 и 4. Пошто је 4 мањи, он се преписује у резултујући низ и увећава се десни показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 _ _ _ _ _
    i           j             k
```

- Сада је $i = 2$ и $j = 2$, па се пореде елементи на позицијама 2 и 2, тј. 6 и 5. Пошто је 5 мањи, он се преписује у резултујући низ и увећава се поново десни показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 _ _ _ _
    i             j             k
```

- Сада је $i = 2$ и $j = 3$, па се пореде елементи на позицијама 2 и 3, тј. 6 и 8. Пошто је 6 мањи, он се преписује у резултујући низ и увећава се леви показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 _ _ _
      i           j             k
```

- Сада је $i = 3$ и $j = 3$, па се пореде елементи на позицијама 3 и 3, тј. 8 и 8. Пошто су једнаки, било који од њих (на пример десни) може бити преписан у резултујући низ и одговарајући показивач се увећава.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 8 _ _
      i             j             k
```

- Пошто у другом низу више нема елемената, преостала два елемента левог низа (8 и 9) се преписују на крај резултата.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 8 8 9
        i           j                 k
```

Сваки показивач пролази кроз један од два низа и укупан број корака је $m+n$, па је сложеност овог алгоритма $O(m + n)$.

```
#include <iostream>

using namespace std;

// objedinjava sortirani niz a od n elemenata i sortirani niz b od m
// elemenata smestajuci rezultat u sortirani niz c
void objedini(int a[], int n, int b[], int m, int c[]) {
  int i = 0, j = 0, k = 0;
  while (i < n && j < m)
    c[k++] = a[i] < b[j] ? a[i++] : b[j++];
  while (i < n)
    c[k++] = a[i++];
  while (j < m)
    c[k++] = b[j++];
}

// najveci broj elemenata niza predvidjen tekstom zadatka

const int MAX = 50000;
int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo prvi niz
  int n;
  cin >> n;
  int a[MAX];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ucitavamo drugi niz
  int m;
  cin >> m;
  int b[MAX];
  for (int i = 0; i < m; i++)
    cin >> b[i];

  // objedinjavamo dva niza u treci
```

```
  int c[MAX + MAX];
  objedini(a, n, b, m, c);

  // ispisujemo rezultat
  for (int i = 0; i < m + n; i++)
    cout << c[i] << " ";
  cout << endl;

  return 0;
}
```

Библиотечка функција

Рецимо и да у стандардној библиотеци језика C++ постоји функција `merge` која врши обједињавање два сортирана низа. Функцији се прослеђују итератори који ограничавају први низ, итератори који ограничавају други низ и итератор на почетак трећег низа у који се смешта резултат. Додатно, могуће је проследити и функцију поређења која одређује редослед сортирања (полазни низови морају бити сортирани у складу са поретком одређеним том функцијом). Задавање поретка посебном функцијом детаљно је описано у задатку Сортирање такмичара.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo prvi niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ucitavamo drugi niz
  int m;
  cin >> m;
  vector<int> b(m);
  for (int i = 0; i < m; i++)
    cin >> b[i];

  // objedinjavamo ih
  vector<int> c(m + n);
  merge(a.begin(), a.end(),
        b.begin(), b.end(),
        c.begin());

  // ispisujemo rezultat
  for (int i = 0; i < c.size(); i++)
    cout << c[i] << " ";
  cout << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Сортирани квадрати бројева

Напиши програм који одређује сортирани низ квадрата сортираног низа бројева.

Улаз: Са стандардног улаза се учитава дужина низа n ($1 \leq n \leq 10^5$), а затим сортиран низ целих бројева између (-10^9 и 10^9), раздвојених размацима.

Излаз: На стандардни излаз исписати сортирани низ квадрата учитаних бројева (бројеви су раздвојени размацима).

Пример 1

Улаз	*Излаз*
`4` `1 3 5 8`	`1 9 25 64`

Пример 2

Улаз	*Излаз*
`6` `-3 -1 0 1 2 5`	`0 1 1 4 9 25`

Пример 3

Улаз	*Излаз*
`3` `-5 -3 -1`	`1 9 25`

Решење

Сортирање

Најједноставнији начин да се задатак реши је да се формира низ квадрата свих учитаних бројева и да се затим тај низ сортиран библиотечком функцијом (као у задатку Сортирање бројева). Потребно је обратити пажњу на то да је за регистровање квадрата бројева потребно користити 64-битни тип података.

Сложеношћу доминира сложеност сортирања библиотечком функцијом, па је укупна сложеност квазилинеарна тј. $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<long long> kvadrati(n);
  for (int i = 0; i < n; i++) {
    long long x;
    cin >> x;
    kvadrati[i] = x * x;
  }

  sort(begin(kvadrati), end(kvadrati));

  cout << kvadrati[0];
  for (int i = 1; i < n; i++)
    cout << " " << kvadrati[i];
  cout << endl;

  return 0;
}
```

Два показивача

Квадрати бројева не морају бити сортирани у истом редоследу као елементи оригиналног низа. Међутим, ако разматрамо само квадрате ненегативних бројева они ће бити сортирани у истом (функција квадрирања је растућа на скупу ненегативних бројева), а ако разматрамо само квадрате негативних бројева они ће бити сортирани у супротном редоследу него оригинални бројеви (функција квадрирања је опадајућа на скупу негативних бројева).

Задатак можемо решити тако што техником два показивача обједињујемо (као у задатку Обједињавање) низ квадрата ненегативних бројева у директном и низ квадрата негативних бројева у обратном редоследу. Пр-

ви ненегативни и последњи негативни број можемо пронаћи линеарном претрагом (пошто је низ сортиран, могуће је применити и бинарну претрагу, али то не би довело до значајно бржег програма).

Сложеност линеарне претраге је $O(n)$, исто колико и сложеност обједињавања, па је укупна сложеност овог алгоритма линеарна тј. $O(n)$.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ucitavamo brojeve
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // rezultujuci niz kvadrata
  vector<long long> kvadrati(n);
  // tekuca slobodna pozicija u rezultujucem nizu
  int k = 0;

  // indeks prvog nenegativnog broja odredjujemo linearnom pretragom
  int nenegativan = 0;
  while (nenegativan < n && a[nenegativan] < 0)
    nenegativan++;
  // indeks poslednjeg negativnog broja
  int negativan = nenegativan - 1;

  // objedinjavamo niz nenegativnih brojeva u direktnom i niz
  // negativnih brojeva u obratnom redosledu

  while (negativan >= 0 && nenegativan < n)
    if (-a[negativan] < a[nenegativan]) {
      kvadrati[k++] = (long long) a[negativan] * a[negativan];
      negativan--;
    } else {
      kvadrati[k++] = (long long) a[nenegativan] * a[nenegativan];
      nenegativan++;
    }

  while (negativan >= 0) {
    kvadrati[k++] = (long long) a[negativan] * a[negativan];
    negativan--;
  }

  while (nenegativan < n) {
    kvadrati[k++] = (long long) a[nenegativan] * a[nenegativan];
    nenegativan++;
  }

  // ispisujemo rezultat
  cout << kvadrati[0];
  for (int i = 1; i < n; i++)
    cout << " " << kvadrati[i];
  cout << endl;
```

```
  return 0;
}
```

Задатак: Заједнички елементи три уређена низа

Три локалне продавнице продају различите производе. Сваки производ је окарактерисан јединственим бар-кодом (природним бројем мањим од милијарду). Ако су познати спискови производа који се продају у те три продавнице (сортирани растуће по бар-кодовима), напиши програм који одређује производе који се продају у све три продавнице.

Улаз: У првој линија стандардног улаза налази се природан број n ($n \leq 50000$) који представља број производа у првој продавници, а у наредној линији n бројева уређених растуће, раздвојених размацима који представљају бар-кодове производа из прве продавнице. Након тога се у истом облику учитавају подаци за другу и подаци за трећу продавницу.

Излаз: На стандардном излазу приказати бар-кодове производа који се продају у свакој од те три продавнице, уређене растући, сваки у посебном реду.

Пример

Улаз	*Излаз*
5 1 5 6 7 10 5 4 5 9 10 17 5 4 5 6 10 13	5 10

Задатак: Близанци

Марија и Петар су близанци и желимо да свакоме од њих двоје купимо по једно одело као поклон за рођендан, али тако да се цене та два поклона што мање разликују (при томе није битно чији поклон ће бити скупљи).

Написати програм који учитава цене свих женских одела и свих мушких одела, а одређује и исписује најмању разлику између цена женског и мушког одела.

Улаз: Опис улаза: са стандардног улаза се учитава:

- у првом реду број мушких одела m ($1 \leq m \leq 50000$),
- у другом реду m целих бројева (цели бројеви између 1 и $2 \cdot 10^9$ раздвојени по једним размаком) - цене мушких одела
- у трећем реду број женских одела z ($1 \leq z \leq 50000$)
- и у четвртом реду z целих бројева (цели бројеви између 1 и $2 \cdot 10^9$ раздвојени по једним размаком) - цене женских одела.

Излаз: На стандардни излаз исписати најмању вредност разлике цена мушког и женског одела.

Пример

Улаз	*Излаз*
5 4680 2120 7940 11530 17820 4 850 13420 5770 6300	1090

Објашњење

Најмања разлика се постиже када се купе одела чије су цене 4680 и 5770 динара.

Задатак: Прости чиниоци 235

Посматрајмо низ бројева чији су прости чиниоци само 2, 3 и 5 (сваки чинилац може да се јави нула и више пута). То су бројеви 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, … Напиши програм који одређује n-ти члан овог низа (бројање креће од 0).

Улаз: Са стандардног улаза се учитава број n ($0 \leq n \leq 10000$).

Излаз: На стандардни излаз исписати тражени n-ти члан низа.

Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
7	9	500	944784

Решење

Директно решење

Директно решење се заснива на израчунавању простих чинилаца сваког броја и провери који од њих има чиниоце само 2, 3 и 5. Функција за проверу може бити веома специјализована: број се дели са 2, са 3 и са 5, докле год је то могуће без остатка и на крају се провери да ли је тиме број сведен на 1. Међутим, ово решење је веома неефикасно.

За проверу чинилаца броја k можемо рећи да се врши у сложености $O(\log n)$. Проверавају се сви бројеви до T_n, где број T_n представља n-ти члан траженог низа, и сложеност је $O(T_n \log T_n)$. Пошто овај низ јако брзо расте (заправо експоненцијално), време експоненцијално зависи од параметра n.

```
#include <iostream>

using namespace std;

bool prosti_cinioci_samo235(long long n) {
  while (n % 2 == 0) n /= 2;
  while (n % 3 == 0) n /= 3;
  while (n % 5 == 0) n /= 5;
  return n == 1;
}

int main() {
  int n;
  cin >> n;
  int k = 0;
  long long i = 0;
  while (k <= n) {
    i++;
    if (prosti_cinioci_samo235(i))
      k++;
  }
  cout << i << endl;
  return 0;
}
```

Обједињавање

Сваки елемент овог низа добија се множењем неког мањег елемента тог низа са 2, 3 или са 5. Замислимо да је низ 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, … већ конструисан. Ако сваки елемент помножимо са 2, добијамо низ 2, 4, 6, 8, 10, 12, 16, 18, … Ако сваки елемент помножимо са 3, добијамо низ 3, 6, 9, 12, 15, 18, 24, 27, 30, … Ако сваки елемент помножимо са 5, добијамо низ 5, 10, 15, 20, 25, 30, 40, … Сви ови низови су сортирани. Приметимо да се обједињавањем ова три низа (тј. проналажењем њихове уније, поступком сличним оном приказаним у задатку Заједнички елементи три уређена низа у ком је тражен пресек дата три низа) добија низ који тражимо. Ово инспирише следећи поступак.

У програм експлицитно чувамо 3 низа у којима се налазе елементи које обједињавамо, док елементе обједињеног низа не морамо чувати – довољно је да чувамо само његов текући елемент који ћемо на крају исписати. Сва три низа можемо алоцирати на n елемената, јер они неће садржати дупликате, мада ће нам бити довољно и мање елемената да би смо стигли до n-тог елемента полазног низа. Уместо низова фиксне величине, можемо употребити и низове који се проширују додавањем елемената на крај. У језику C++ то може бити вектор.

На почетку у први постављамо само број 2, у други број 3, а у трећи број 5. Помоћу три показивача памтимо

позицију текућег елемента у сваком од та три низа. Узимамо минимални од та три елемента и уклањамо га са почетка (пошто се он може истовремено јавити у више низова, увећавамо показиваче у свим низовима док год им је он на почетку). Множимо га редом, са 2, 3 и 5 и додајемо резултате на крај одговарајућих низова. Поступак настављамо све док са почетка низова не издвојимо n елемената.

Сваки елемент се може највише једном јавити у сва три низа. У процесу се анализирају само елементи траженог низа T_n и то сваки од њих највише 3 пута. Сложеност, дакле, линеарно зависи од n и износи $O(n)$. Меморијска сложеност је такође линеарна у односу на n и износи $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<long long> a2, a3, a5;
  a2.push_back(2);
  a3.push_back(3);
  a5.push_back(5);
  long long t = 1;
  int i2 = 0, i3 = 0, i5 = 0;
  for (int i = 0; i < n; i++) {
    t = min({a2[i2], a3[i3], a5[i5]});
    a2.push_back(2*t);
    a3.push_back(3*t);
    a5.push_back(5*t);
    while (i2 < a2.size() && a2[i2] == t) i2++;
    while (i3 < a3.size() && a3[i3] == t) i3++;
    while (i5 < a5.size() && a5[i5] == t) i5++;
  }
  cout << t << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Тастатура и миш

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Двобојка

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Трансформације низа “у месту” - два показивача

Постоји неколико начина да се низ “у месту” трансформише само једним проласком кроз низ (ти приступи имају линеарну временску сложеност). Приказаћемо неколико могућности, све засноване на техници два показивача.

Парни лево, непарни десно - размена наопаких

Претпоставићемо да су у сваком кораку петље познати индекси l и d тако да су елементи низа груписани тако да су:

- сви елементи на позицијама из интервала $[0, l)$ парни,
- сви елементи на позицијама из интервала $[l, d]$ још непрегледани и
- сви елементи на позицијама из интервала (d, n) непарни.

Инваријанта је, дакле, да је распоред елемената у низу облика `ppp???nnn`, где су `l` и `d` позиција првог тј. последњег непознатог елемента (обележеног упитником).

На почетку иницијализујемо l на нулу, а d на $n-1$ (тада су интервали $[0, l)$ и (d, n) празни, а сви елементи у интервалу $[l, d] = [0, n-1]$ су још непрегледани). Петљу у приниципу извршавамо док још има непрегледаних елемената тј. док је $l \leq d$, међутим, у овом задатку можемо је завршити и корак раније. Петља се извршава док је $l < d$ и завршава када је $l = d$, јер какав год да је тај последњи непрегледани елемент на позицији $l = d$, он ће се моћи припојити или левом или десном делу низа и неће бити потребе премештати га.

У телу петље радимо следеће:

- Прво испитујемо да ли је елемент на позицији l паран и ако јесте, онда само увећавамо вредност l за 1.
- У супротном, проверавамо да ли је елемент на позицији d непаран и ако јесте, онда само умањујемо вредност d за 1.
- На крају, ако ниједна од претходне две провере није успела, знамо да је елемент на позицији l непаран, а елемент на позицији d паран, размењујемо их, увећавамо l за 1 и умањујемо d за 1.

Када се петља заврши, елементи су у жељеном редоследу.

Прикажимо рад алгоритма на једном примеру.

```
l                 d
3  8  7  4  5  1  6  2

   l              d
2  8  7  4  5  1  6  3

      l           d
2  8  7  4  5  1  6  3

         l     d
2  8  6  4  5  1  7  3

            l  d
2  8  6  4  5  1  7  3

            ld
2  8  6  4  5  1  7  3
```

Променљива `l` се увећава, а променљива `d` се умањује све док се не сусретне, што се догађа у тачно n корака, па је укупна сложеност алгоритма $O(n)$.

Докажимо и формално коректност претходног алгоритма. Током извршавања важи раније описана инваријанта, а важи и да је $0 \leq l \leq d + 1 \leq n$.

Иницијализацијом вредности l на нулу, а вредности d на $n-1$ постижемо да су ови услови на почетку задовољени (јер су сви елементи у интервалу $[l, d] = [0, n-1]$ још непрегледани).

У телу петље радимо следеће:

- Прво испитујемо да ли је елемент на позицији l паран и ако јесте, онда само увећавамо вредност l за 1 чиме наметнута инваријанта остаје да важи. Заиста, парни су били сви елементи из интервала $[0, l)$, паран је и елемент на позицији l, па су парни сви елементи из интервала $[0, l] = [0, l')$, где је $l' = l + 1$, нова вредност променљиве l. Пошто је $d' = d$, елементи на позицијама из интервала (d', n) су непарни.
- У супротном, проверавамо да ли је елемент на позицији d непаран и ако јесте, онда само умањујемо вредност d за 1 чиме наметнута инваријанта опет остају на снази (аргументација је слична оној у претходном случају).

- На крају, ако ниједна од претходне две провере није успела, знамо да је елемент на позицији l непаран, а елемент на позицији d паран, размењујемо их, увећавамо l за 1 и умањујемо d за 1 чиме инваријанта остаје да важи. Заиста, важи да је $l' = l + 1$ и $d' = d - 1$. Сви елементи из интервала позиција $[0, l)$ су парни, а паран је елемент на позицији l (јер је разменом паран елемент са позиције d доведен на позицију l). Зато су парни сви елементи у интервалу $[0, l')$. Слично, пошто су сви елементи у интервалу позиција (d, n) били непарни, а пошто је након размене непарни елемент са позиције l доведен на позицију d, непарни су и сви елементи из интевала $[0, d')$.

Када се петља заврши, распоред је коректан. Наиме, пошто на крају петље услов $l < d$ није испуњен, а пошто је $l \leq d + 1$, тада је $l = d$ или је $l = d + 1$.

- Ако је $l = d + 1$, знамо да је низ разбијен на сегмент парних елемената на позицијама $[0, l)$ и непарних на позицијама $(d, n) = [d + 1, n) = [l, n)$.
- Размотримо случај $l = d$.
  - Ако је елемент на позицији l паран, пошто су на основу инваријанте парни и сви елементи на позицијама $[0, l)$, знамо да ће парни бити сви елементи на позицијама $[0, l]$, док ће елементи на позицијама $(d, n) = [l + 1, n)$ бити непарни.
  - Слично, ако је елемент на позицији l непаран, тада су парни сви елементи на позицијама $[0, l)$, а непарни су сви елементи на позицијама $[l, n)$ (јер на основу инваријанте знамо да су још непарни и елементи на позицијама $(d, n) = (l, n)$).

  У оба случаја је, дакле, постигнут жељени распоред елемената.

Приметимо да смо прекидом петље када је $l = d$, уштедели једну итерацију петље (што није нарочито значајно), али смо закомпликовали доказ коректности, па се природно поставља питање колико је та оптимизација имала смисла.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  // ubrzavamo ucitavanje i ispis
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // odrzavamo uslov
  // [0, l) - parni
  // (d, n) - neparni
  // [l, d] - nepoznati

  // u pocetku su svi nepoznati
  int l = 0, d = n-1;
  // dok jos ima nepoznatih elemenata
  while (l < d) {
    // ako je na mestu l paran, ostavljamo ga na svom mestu i pomeramo
    // se na naredni element
    if (a[l] % 2 == 0)
      l++;
    // ako je na mestu d neparan, ostavljamo ga na svom mestu i
    // pomeramo se na prethodni element
    else if (a[d] % 2 != 0)
```

```
      d--;
    else
      // na mestu l je neparan, a na mestu d je paran broj, pa ih
      // razmenjujemo i pomeramo se po oba kraja
      swap(a[l++], a[d--]);
  }

  // ispisujemo rezultat
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;
}
```

Парни лево, непарни десно - проналажење и размена наопаких угнежђеним петљама

Често се у оквиру технике два показивача користе угнежђене петље које померају те показиваче све до елемента који треба на неки начин обрадити. У овом задатку можемо у унутрашњим петљама пронаћи два елемента која стоје наопако и онда их разменити.

Иако се у програму користе угнежђене петље, сложеност је линеарна тј. $O(n)$. Наиме укупан број промена показивача l и d је највише n (оба се померају у једном смеру, док се не сусретну).

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  // ubrzavamo ucitavanje i ispis
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // odrzavamo uslov
  // [0, l) - parni
  // (d, n) - neparni
  // [l, d] - nepoznati


  // u pocetku su svi nepoznati
  int l = 0, d = n-1;
  // dok jos ima nepoznatih elemenata
  while (l < d) {
    // dok je na mestu l paran, ostavljamo ga na svom mestu i pomeramo
    // se na naredni element
    while (l < d && a[l] % 2 == 0)
      l++;
    // dok je na mestu d neparan, ostavljamo ga na svom mestu i
    // pomeramo se na prethodni element
    while (l < d && a[d] % 2 != 0)
      d--;

    if (l < d)
      // na mestu l je neparan, a na mestu d je paran broj, pa ih
      // razmenjujemo i pomeramo se po oba kraja
      swap(a[l++], a[d--]);
```

```
  }

  // ispisujemo rezultat
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;
}
```

Парни лево, непарни десно - размена са непознатим десно

У другом решењу опет претпостављамо да имамо променљиве l и d и да важе исти услови као у претходном решењу. Самим тим и иницијализација променљивих l и d је идентична. Разлика је, међутим, у телу петље. Проверавамо да ли је елемент на позицији l паран и ако јесте, онда само увећавамо вредност l. У супротном се елемент на позицији l размењује са елементом на позицији d и позиција d се умањује за 1, док позиција l остаје непромењена (десни крај низа у којем су непарни елементи проширујемо за један непарни елемент, док леви крај низа остаје непромењен - пошто се l не мења, непознати елемент који је доведен на место l ће бити размотрен у следећем кораку петље).

Показивачи l и d се поново померају са два краја низа, док се не сусретну, што ће се десити у највише n корака, па је сложеност $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  // ubrzavamo ucitavanje i ispis
  ios_base::sync_with_stdio(false);

  // ucitavamo elemente niza
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // odrzavamo naredni uslov
  // [0, l) - parni
  // [l, d] - nepoznati
  // (d, n) - neparni

  // na pocetku su svi nepoznati
  int l = 0, d = n-1;

  // dok ima nepoznatih
  while (l < d) {
    // paran element ostaje na svom mestu
    if (a[l] % 2 == 0)
      l++;
    else
      // neparan se razmenjuje sa poslednjim nepoznatim
      swap(a[l], a[d--]);
  }

  // ispisujemo rezultat
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;
}
```

Парни, непарни, па непознати

У овом решењу инваријанта је мало другачија. Памтимо индекс k и текући индекс i и претпостављамо да су

- елементи на позицијама из интервала $[0, k)$ парни,
- елементи из интервала $[k, i)$ непарни,
- елементи из интервала $[i, n)$ још непрегледани.

Дакле, намећемо инваријанту да је распоред облика `pppnnn???`, где је i позиција првог непознатог, а k позиција првог непарног елемената.

Размотримо како да из инваријанте закључимо како треба иницијализовати промењиве. Пошто су сви елементи из интервала $[i, n)$ непрегледани, промењиву i морамо инцијализовати на 0. Пошто су сви елементи из интервала $[0, k)$ парни, а $[k, i)$ непарни, и k морамо поставити на 0.

Инваријанта јасно диктира и услов петље. Наиме, петља се извршава док још има непрегледаних елемената тј. док је $i < n$. У сваком кораку петље i се увећава за 1 (користимо класичну бројачку петљу `for` по промењивој i), чиме се сужава интервал непрегледаних елемената.

Размотримо како треба да изгледа тело петље да би инваријанта остала испуњена.

- Ако је елемент на текућој позицији i паран, онда га размењујемо са првим непарним елементом, а то је елемент на позицији k. Изузетак је случај када је $k = i$, када заправо не долази до размене (елемент се мења сам са собом). У оба случаја се k увећава за 1.
- У супротном, елемент на позицији i је непаран он остаје на свом месту и у телу петље није потребно ништа урадити (грану `else` у коду није потребно наводити).

Прикажимо рад алгоритма на једном примеру.

```
k                   n
3  8  7  4  5  1  6  2
i

k                   n
3  8  7  4  5  1  6  2
   i

   k                n
8  3  7  4  5  1  6  2
      i

   k                n
8  3  7  4  5  1  6  2
         i

      k             n
8  4  7  3  5  1  6  2
            i

      k             n
8  4  7  3  5  1  6  2
               i

      k             n
8  4  7  3  5  1  6  2
                  i

         k          n
8  4  6  3  5  1  7  2
                     i

            k       n
8  4  6  2  5  1  7  3
                     i
```

У програму се користи класична бројачка петља `for` која се завршава у n корака, па је укупна сложеност алгоритма $O(n)$.

Докажимо и формално коректност претходног поступка. Уз описану инваријанту важи и да је $0 \leq k \leq i \leq n$.

Пошто је на почетку $i = k = 0$, важи да су сви елементи у интервалу $[i, n) = [0, n)$ непрегледани. Интервали $[0, k) = [k, i) = [0, 0)$ су празни, па задовољавају услове, а тривијално Важи и да је $0 \leq k \leq i \leq n$.

У телу петље се врше следеће акције.

- Ако је елемент на текућој позицији i паран, онда га размењујемо са првим непарним елементом, а то је елемент на позицији k. Изузетак је случај када је $k = i$, када заправо не долази до размене (елемент се мења сам са собом). У оба случаја се k увећава за 1. Дакле, на основу инваријанте знамо да су елементи на позицијама $[0, k)$ парни, да је након размене елемент на позицији k паран, па су парни и сви елементи на позицијама $[0, k') = [0, k + 1)$. Елементи на позицијама $[k', i') = [k + 1, i + 1)$ су непарни. Наиме, ако је $k = i$, овај интервал је празан, а ако је $k < i$, тада је пре размене елемент на позицији k био непаран (јер на основу инваријанте знамо да су сви елементи на позицијама $[k, i)$ били непарни, па самим тим и елемент на позицији k, који је сада доведен на позицију i).
- У супротном, елемент на позицији i је непаран он остаје на свом месту. Пошто се у телу петље не дешава ништа, важи $k' = k$ и $i' = i$, а инваријанта прилично очигледно остаје на снази.

По завршетку петље услов $i < n$ није испуњен, па пошто на основу инваријанте важи $0 \leq k \leq i \leq n$, важи да је $i = n$. Зато је интервал непознатих $[i, n)$ празан, сви елементи из интервала $[0, k)$ су парни, из интервала $[k, i) = [k, n)$ су непарни и постигнут је тражени распоред.

Заустављање се једноставно доказује јер се у сваком кораку сужава интервал непознатих $[i, n)$.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  // ubrzavamo ucitavanje i ispis
  ios_base::sync_with_stdio(false);

  // ucitavamo elemente niza
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // odrzavamo uslov
  // [0, k) - parni
  // [k, i) - neparni
  // [i, n) - nepoznati

  // u pocetku su svi elementi nepoznati
  int k = 0;
  for (int i = 0; i < n; i++)
    // ako je element paran razmenjujemo ga sa prvim neparnim
    if (a[i] % 2 == 0)
      swap(a[i], a[k++]);
    // a ako je neparan, ne pomeramo ga

  // ispisujemo rezultat
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;
}
```

Коришћење помоћног низа

Задатак се може решити тако што ће се кроз улазни низ a ићи истовремено слева и здесна и у низ b при кретању слева копирати парни елементи низа a, а при кретању здесна непарни елементи низа a. Ово решење не поштује поступак који се тражио у тексту задатка, па представља одређени вид варања.

Иако је меморијска сложеност и даље $O(n)$, у овом програму се непотребно користи додатни низ. Временска сложеност је линеарна тј. $O(n)$, што је оптимално.

```
#include <iostream>
#include <vector>
using namespace std;


int main() {
  // ubrzavamo ucitavanje i ispis
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // pomocni niz
  vector<int> b(n);
  for (int i = 0, j = n - 1, k = 0; k < n; k++) {
    // kopira parne elemente niza A u niz B sleva nadesno
      if (a[k] % 2 == 0)
        b[i++] = a[k];
      // kopira neparne elemente niza A u niz B sdesna ulevo
      if (a[n - 1 - k] % 2 == 1)
        b[j--] = a[n - 1 - k];
  }

  // ispisujemo elemente niza B
  for (int i = 0; i < n; i++)
    cout << b[i] << endl;
}
```

Прилагођено учитавање

Задатак можемо решити и тако што користимо "трик" да се елементи већ приликом учитавања у низ слажу на одговарајући начин. Тако одржавамо индексе l и d које иницијализујемо на 0 тј. $n-1$ и парне елементе увећавамо на позицију l увећавајући након тога l за 1, док непарне елементе учитавамо на позицију d умањујући након тога d за 1. Ово решење не поштује поступак који се тражио у тексту задатка, па представља одређени вид варања.

Временска сложеност је $O(n)$, меморијска такође $O(n)$ и не користе се помоћни низови.

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  // ubrzavamo ucitavanje i ispis
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
```

```
  cin >> n;
  vector<int> a(n);
  for (int i = 0, l = 0, d = n-1; i < n; i++) {
    int x;
    cin >> x;
    // parne brojeve ucitavamo na levi kraj niza a
    if (x % 2 == 0)
      a[l++] = x;
    // neparne brojeve ucitavamo na desni kraj niza a
    else
      a[d--] = x;
  }

  // ispisujemo elemente niza a
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;
}
```

Задатак: Тробојка

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Два пролаза кроз низ

Један приступ је да се до решења дође у две фазе. У првој фази би се на почетак низа довели сви елементи који су мањи од броја A, а иза њих би се поставили сви елементи који су већи или једнаки броју A. Ово је могуће урадити неким од алгоритама које смо описали у задатку Двобојка. Након тога, у другој фази обрађује се само део низа, и он се поново истим поступком дели на елементе који су мањи или једнаки од броја B (то ће бити тачно елементи из интервала $[A, B]$) и елементе који су већи од B. Поделу можемо реализовати засебном функцијом, која прима део низа који се реорганизује и границу на основу које се врши подела, а која враћа позицију на којој почиње други део реорганизованог низа.

Програм се решава у две фазе и свака је сложености $O(n)$. Укупна сложеност овог приступа је, дакле, $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// ucitavanje elemenata niza
vector<int> unosNiza() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  return a;
}


// organizuje elemente niza na pozicijama iz intervala [levo, desno]
// tako da vazi da su za neku poziciju k u intervalu [levo, k) svi
// elementi iz intervala (-Inf, X], dok su u intervalu [k, desno]
// svi elementi iz intervala (X, Inf)
// funkcija vraca poziciju k
int podeli(vector<int>& a, int levo, int desno, int X) {
```

```
  int k = levo;
  for(int i = levo; i <= desno; i++) {
    if (a[i] <= X) {
      swap(a[i], a[k]);
      k++;
    }
  }
  return k;
}

// ispis elemenata niza na standardni izlaz
void ispisNiza(const vector<int>& a, int n) {
  for(int i = 0; i < n; i++)
    cout << a[i] << " ";
  cout << endl;
}


int main()  {
  // ucitavamo niz
  // ucitavamo elemente niza
  vector<int> a = unosNiza();
  int n = a.size();

  // ucitavamo interval podele
  int A, B;
  cin >> A >> B;

  // delimo niz tako da se u intervalu pozicija [0, krajA) nalaze svi
  // elementi niza koji su iz intervala (-Inf, A - 1] = (-Inf, A), a u
  // intervalu pozicija [krajA, n) nalaze elementi iz intervala [A, +Inf)
  int krajA = podeli(a, 0, n - 1, A - 1);

  // delimo deo niza iz intervala pozicija [krajA, n) tako da se u
  // intervalu pozicija [krajA, krajB) nalaze svi elementi iz
  // intervala [A, B], a u intervalu pozicija [krajB, n) svi elementi
  // iz intervala (B, +Inf)
  podeli(a, krajA, n - 1, B);

  // ispisujemo rezultat
  ispisNiza(a, n);

  return 0;
}
```

Један пролаз кроз низ

Задатак можемо решити помоћу само једног пролаза кроз низ и то "у месту" тј. без коришћења помоћног низа. Алгоритам у наставку познат је под називом "Холандска застава тробојка" (енгл. Dutch national flag) и приписује се чувеном информатичару Дајкстри (енгл. Edsger W. Dijkstra).

Одржаваћемо три променљиве l, d и i и током петље наметнућемо да важи $0 \leq l \leq d \leq i \leq n$ и да важе следећи услови.

- У интервалу позиција $[0, l)$ налазиће се елементи мањи од A тј. бројеви из интервала $(-\infty, A)$,
- у интервалу позиција $[l, i)$ налазиће се елементи из интервала $[A, B]$,
- у интервалу позиција $[i, d)$ налазиће се елементи који још нису испитани,
- у интервалу позиција $[d, n)$ налазиће се елементи који су већи од B тј. елементи из интервала $(B, +\infty)$.

Дакле, одржавамо распоред `<<<===???>>>`, где су са `<` обележени елементи прве групе, са `=` елементи друге,

а са > елементи треће групе.

Да би инваријанта важила пре уласка у петљу, јасно је да мора да важи да је $i = 0$ и $d = n$ (јер су сви елементи из интервала $[i, d) = [0, n)$ неиспитани). Такође, да бисмо били сигурни да су и интервалу $[0, l)$ сви елементи мањи од A, тај интервал мора бити празан и мора да важи да је $l = 0$. Након овакве иницијализације и интервал $[l, i) = [0, 0)$ и интервал $[d, n) = [n, n)$ је празан, па задовољава наметнути услов.

Петља ће се извршавати док год има неиспитаних елемената, а то је док је $i < d$. Размотримо како треба да изгледа тело петље, да би услови били одржани.

- Ако је елемент на позицији i мањи од броја A тада ћемо га заменити са елементом на позицији l (првим елементом из интервала $[A, B]$), након чега можемо увећати и i и l.
- У супротном, ако је елемент на позицији i мањи или једнак од B он припада интервалу $[A, B]$ и већ је на свом допуштеном месту, па само можемо увећати вредност i.
- У супротном елемент је већи од B и тада можемо смањити вредност d и разменити елемент на позицији i са елементом на (умањеној) позицији d, не мењајући вредност i (да би се елемент који је управо доведен на позицију i могао испитати у наредној итерацији).

На крају петље важи да је $i = d$. Уз остале наметнуте услове тврђење одатле следи (елементи из интервала позиција $[0, l)$ су мањи од A, елементи из интервала позиција $[l, i) = [l, d)$ су између A и B, интервал непрегледаних елемената $[i, d)$ је празан, док су елементи из интервала $[d, n)$ већи од B. Дакле, низ је разбијен на надовезане сегменте $[0, l)$, $[l, d)$ и $[d, n)$ и у сваком сегменту се налазе одговарајући елементи.

Размотримо рад алгоритма на једном примеру. Нека је $A = 4$, $B = 7$ и нека низ има садржај `5 1 8 6 3 9 4 2`. У наставку ћемо приказати стање низа током извођења алгоритма.

```
l               d
5 1 8 6 3 9 4 2
i

l               d
5 1 8 6 3 9 4 2
  i

  l             d
1 5 8 6 3 9 4 2
    i

  l           d
1 5 2 6 3 9 4 8
    i

    l         d
1 2 5 6 3 9 4 8
      i

    l         d
1 2 5 6 3 9 4 8
        i

      l       d
1 2 3 6 5 9 4 8
          i

      l     d
1 2 3 6 5 4 9 8
          i

      l     d
1 2 3 6 5 4 9 8
```

```
          i
```

У сваком кораку петље се или увећава `i` или смањује `d`, док се не сусретну, што се дешава у n корака. Сложеност овог приступа је, дакле, $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// funkcija ucitava elemente u vektor
vector<int> unosNiza() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  return a;
}

// funkcija organizuje elemente vektora tako da se prvo nalaze elementi
// za koje vazi da su iz intervala (-Inf, A), nakon toga dolaze
// elementi iz intervala [A, B], i nakon toga elementi iz intervala
// (B, Inf)
void podelaNiza(vector<int>& niz, int A, int B) {
  // - u intervalu pozicija [0, l) su elementi iz intervala (-Inf, A)
  // - u intervalu pozicija [l, i) su elementi iz intervala [A, B]
  // - u intervalu pozicija [i, d) su jos neispitani elementi
  // - u intervalu pozicija [d, n) su elementi iz intervala (B, Inf)
  int l = 0, i = 0, d = niz.size();
  // dok god postoje neispitani elementi
  while (i < d) {
    if (niz[i] < A)
      // menjamo tekuci element sa prvim elementom iz intervala [A, B]
      swap (niz[i++], niz[l++]);
    else if (niz[i] <= B)
      // tekuci element ostaje na svom mestu
      i++;
    else
      // menjamo tekuci element sa poslednjim neispitanim
      swap(niz[i], niz[--d]);
  }
}

// ispis elemenata vektora na standardni izlaz
void ispisNiza(const vector<int>& a, int A, int B) {
  int i = 0;
  // ispisujemo elemente iz intervala (-Inf, A)
  while (i < a.size() && a[i] < A)
    cout << a[i++] << " ";
  cout << endl;
  // ispisujemo elemente iz intervala [A, B]
  while (i < a.size() && a[i] <= B)
    cout << a[i++] << " ";
  cout << endl;
  // ispisujemo elemente iz intervala (B, +Inf)
  while (i < a.size())
    cout << a[i++] << " ";
```

```
  cout << endl;
}


int main()  {
  // ucitavamo elemente niza
  vector<int> a = unosNiza();
  // ucitavamo interval [A, B]
  int A, B;
  cin >> A >> B;
  // reorganizujemo elemente po intervalima (-inf, A), [A, B] i [B, inf)
  podelaNiza(a, A, B);
  // ispisujemo elemente niza
  ispisNiza(a, A, B);
  return 0;
}
```

Задатак: Оптимални сервис

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Број парова датог збира

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Види другачија решења овог задатка.

Задатак: Тројке датог збира (3sum)

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Разлика висина

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Груба сила

Наиван начин да се задатак реши је да се испитају сви уређени парови ученика и да се преброје они чија је разлика једнака траженој.

Пошто уређених парова ученика има n^2, сложеност оваквог алгоритма је $O(n^2)$.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int razlika;
  cin >> razlika;
  int n;
  cin >> n;
  vector<int> a(n);
```

```
  for (int i = 0; i < n; i++)
    cin >> a[i];
  int broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      if (a[j] - a[i] == razlika)
        broj++;
  cout << broj << endl;
  return 0;
}
```

Сортирање

Као и у многим проблемима претраге, сортирање низа може довести до ефикаснијих решења. За почетак, ако је низ сортиран, довољно је само да проверавамо парове такве да је други елемент пара иза првог. Дакле, за сваки елемент низа одређујемо број елемената иза њега који са њим дају тражену разлику (он је умањилац, а тражимо потенцијалне умањенике). Пошто је низ сортиран, чим наиђемо на први елемент иза њега који има већу разлику од тражене, такви ће бити и сви елементи у наставку низа, па можемо извршити одсецање и прећи на обраду следећег елемента (умањиоца).

У најгорем случају не долази до одсецања, а проверавају се сви парови којих има $\binom{n}{2} = \frac{n(n-1)}{2}$, па је сложеност овог приступа $O(n^2)$. Наравно, у сложеност је укључена и сложеност сортирања $O(n \log n)$, међутим, она је у овом случају занемарива у односу на сложеност испитивања свих парова.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int razlika;
  cin >> razlika;
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  sort(begin(a), end(a));

  int broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      if (a[j] - a[i] == razlika)
        broj++;
      else if (a[j] - a[i] > razlika)
        break;

  cout << broj << endl;

  return 0;
}
```

Техника два показивача

Задатак можемо решити техником два показивача. Релативно слична техника је коришћена у решењу задатка Број парова датог збира. Низ мора бити сортиран, а оба показивача се крећу слева надесно. Једноставности ради, претпоставимо прво да у низу нема дупликата.

Прикажимо како би алгоритам радио на примеру одређивања броја елемената чија је разлика 8 у наредном низу.

```
1 3 7 8 11 14 16 30 38
```

- Крећемо од пара `1 3`. Пошто је разлика мања од тражене елемент `3` не може бити умањеник, па повећавамо умањеник на `7`.
- Анализирамо пар `1 7`. Ситуација је опет иста, па опет повећавамо умањеник на `8`. Наиме, померањем умањиоца са `1` на `3`, разлика би се само смањила, па `7` заиста не може бити умањеник.
- Анализирамо пар `1 8`. И ту је ситуација иста, па опет повећавамо умањеник на `11` (поново закључујемо да се померањем умањиоца са `1` надесно, на `3` или `7` разлика смањује, па `8` заиста не може бити умањеник).
- Анализирамо пар `1 11`. Овај пут је разлика већа од тражене. Стога можемо закључити да `1` не може бити умањилац (даљим померањем умањеника надесно, разлика би се само повећала). Стога прелазимо на наредни умањилац, а то је `3`. Кључна напомена је да су разлике свих умањеника испред `11` и броја `3` мање од тражене разлике `8` (јер су такве биле разике и када је умањилац био мањи)
- Анализирамо пар `3 11`. То је први пар који има дату разлику. Ако су елементи различити, тада се за све умањенике после `11` добија већа разлика у односу на умањилац `3`, па можемо да померимо умањилац на `7`. Слично, умањеник `11` не може да направи ни један даљи пар чија би разлика била једнака траженој, па можемо да померимо умањеник на `14`.
- Анализирамо пар `7 14`. Разлика је мања од тражене, па повећавамо умањеник на `16`.
- Анализирамо пар `7 16` разлика је већа од тражене, па померамо умањилац на `8`.
- Анализирамо пар `8 16` чија је разлика једнака траженој. Након тога можемо повећати и умањилац на `11` и умањеник на `17`.
- Анализирамо пар `11 30` и пошто му је разлика већа од тражене, померамо умањилац на `14`.
- Анализирамо пар `14 30` и пошто му је разлика већа од тражене, померамо умањилац на `16`.
- Анализирамо пар `16 30` и пошто му је разлика већа од тражене, померамо умањилац на `30`.
- Анализирамо пар `30 30` коме је разлика мања од тражене, па померамо умањеник на `38`.
- Анализирамо пар `38 30` коме је разлика једнака траженој, па померамо и умањилац и умањеник. Пошто не постоји већи умањеник, алгоритам се завршава.

Опишимо формално претходни поступак и докажимо његову коректност. Одредимо колико парова дате разлике r постоји у интервалу $[i, n)$, ако знамо да важи инваријанта да је је $a_{j-1} - a_i < r$. Вршимо иницијализацију $i = 0$ и $j = 1$, тако да одређујемо број парова и интервалу $[0, n)$, а инваријанта је задовољена јер је $a_{j-1} - a_i = a_0 - a_0 = 0 < r$.

- Ако је $j = n$, тада у интервалу $[i, n)$ не постоји ни један пар дате разлике. Заиста, на основу инваријанте важи да је $a_{n-1} - a_i < r$. Пошто је низ сортиран, и повећањем i и смањивањем j разлика се смањује. Зато парови бројева унутар интервала $[i, n)$ имају мању разлику од r.
- Ако је $a_j - a_i < r$, тада знамо да у интервалу $[i, j]$ не постоји ни један пар бројева чија је разлика једнака r (јер је разлика елемената унутар интервала увек мања неко разлика крајњих елемената). Инваријанта је задовољена за пар $(i, j + 1)$ па увећавамо j и настављамо поступак.
- Ако је $a_j - a_i > r$, тада ни један пар $a_{j'} - a_i$ за $i < j' < n$ нема разлику r. На основу инваријанте знамо да је $a_{j-1} - a_i$ мање од r, па пошто је низ сортиран то важи и за све елементе $i < j' < j$. Пошто је $a_j - a_i > r$ и пошто је низ сортиран повећањем j се повећава разлика, па су разлике за $j \leq j' < n$ веће од r. Зато се у интервалу $[i, n)$ сви евентуални парови чија је разлика r налазе у интервалу $[i + 1, n)$ и поступак настављамо тако што увећавамо i за 1. Још морамо доказати да тада инваријанта важи тј. да је $a_{j-1} - a_{i+1} < r$, међутим то важи јер је низ сортиран и важи $a_i < a_{i+1}$, а на основу инваријанте је важило да је $a_{j-1} - a_i < r$.
- На крају, ако је $a_j - a_i = r$, тада смо пронашли један пар. Пошто смо претпоставили да у низу нема дупликата и да је низ сортиран, a_i не може бити члан ни једног другог пара са разликом r у интервалу $[i, n)$ - пошто је низ сортиран померањем умањеника налево разлика се смањује, а померањем надесно,

она се повећава. Дакле, сви евентуални парови чија је разлика r налазе се у интервалу $[i+1, n)$. Поступак се може наставити увећавањем и индекса i и индекса j за 1. Заиста, инваријанта је задовољена јер је $a_{j+1-1} - a_{i+1} = a_j - a_{i+1} < a_j - a_i = r$.

Сложеност овог приступа је $O(n \log n)$ захваљујући почетном сортирању, док је сложеност друге фазе, након сортирања линеарна тј. $O(n)$. Заиста и умањеник и умањилац се крећу у истом смеру (вредност оба показивача се само увећава), па се може направити највише $2n$ корака.

Пређимо сада на случај у ком се елементи у низу могу понављати.

Обрада дупликата читањем серије истих елемената

Ако се елементи у низу понављају, онда у тренутку када нађемо први пар (i, j) такав да је $a_i - a_j = r$, одређујемо број појављивања n_i елемента a_i и број појављивања n_j елемента a_j, број парова увећавамо за $n_i \cdot n_j$ (јер свако појављивање вредности a_i можемо искомбиновати са сваким појављивањем вредности a_j) и након тога поступак настављамо од индекса $(i + n_i, j + n_j)$.

Сложеношћу и даље доминира сортирање чија је сложеност $O(n \log n)$, док је сложеност друге фазе и даље линеарна тј. $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  int razlika;
  cin >> razlika;
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  sort(begin(a), end(a));

  int broj = 0;
  int i = 0, j = 1;
  while (j < n) {
    if (a[j] - a[i] < razlika)
      j++;
    else if (a[j] - a[i] > razlika)
      i++;
    else {
      // pronalazimo sve elemente jednake a[i]
      int ii;
      for (ii = i+1; ii < n && a[ii] == a[i]; ii++)
        ;
      // odredjujemo koliko ih ima
      int broj_ai = ii - i;
      // preskacemo ih
      i = ii;

      // pronalazimo sve elemente jednake a[j]
      int jj;
      for (jj = j+1; jj < n && a[jj] == a[j]; jj++)
        ;
      // odredjujemo koliko ih ima
      int broj_aj = jj - j;
```

```
      // preskacemo ih
      j = jj;

      // uvecavamo brojac za broj parova (a[i], a[j])
      broj += broj_ai * broj_aj;
    }
  }

  cout << broj << endl;

  return 0;
}
```

Обрада дупликата пребројавањем појављивања

Још један начин да се реши проблем понављања елемената је да се у првој фази низ вредности трансформише у низ парова који садрже вредности и њихов број појављивања.

Сортирање је сложености $O(n \log n)$. Пребројавање елемената се након тога извршава у сложености $O(n)$, једним проласком кроз низ, након чега се са два показивача пролази кроз низ опет у сложености $O(n)$. Укупна сложеност је, дакле, $O(n \log n)$. Имплементација користи и помоћни низ, али меморијска сложеност остаје $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  // ucitavamo podatke
  ios_base::sync_with_stdio(false);
  int razlika;
  cin >> razlika;
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // sortiramo niz
  sort(begin(a), end(a));

  // odredjujemo broj pojavljivanja svakog elementa
  vector<pair<int, int>> b;
  b.reserve(n);
  b.push_back(make_pair(a[0], 1));
  for (int i = 1; i < n; i++) {
    if (a[i] == b.back().first)
      b.back().second++;
    else
      b.push_back(make_pair(a[i], 1));
  }

  // trazimo elemente cija je razlika jednaka datoj
  int broj = 0;
  int i = 0, j = 0;
  while (j < b.size()) {
    if (b[j].first - b[i].first < razlika)
      j++;
```

```
    else if (b[j].first - b[i].first > razlika)
      i++;
    else {
      broj += b[j].second * b[i].second;
      i++; j++;
    }
  }
  cout << broj << endl;
  return 0;
}
```

Обрада дупликата коришћењем мапе тј. речника

Још један интересантан начин имплементације је да уместо сортирања низа употребимо мапу тј. речник која пресликава елементе у њихов број појављивања и да затим са два показивача пролазимо кроз кључеве мапе (у њиховом сортираном редоследу).

Као имплементацију сортиране мапе језику C++ можемо употребити колекцију `map`.

Пошто је сложеност појединачног уметања и појединачне претраге у сортираној мапи тј. речнику $O(\log n)$, овим добијамо алгоритам сложености $O(n \log n)$. Итерација кроз мапу помоћу два показивача је сложености $O(n)$.

```
#include <iostream>
#include <map>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int razlika;
  cin >> razlika;

  // preslikavamo svaki element u njegov broj pojavljivanja
  map<int, int> m;
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    m[x]++;
  }

  // odredjujemo broj elemenata cija je razlika jednaka datoj
  int broj = 0;
  auto i = m.begin(), j = m.begin();
  while (j != m.end()) {
    if (j->first - i->first < razlika)
      j++;
    else if (j->first - i->first > razlika)
      i++;
    else {
      broj += j->second * i->second;
      j++;
    }
  }
  cout << broj << endl;
  return 0;
}
```

Задатак: Сегмент датог збира у низу природних бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Пуно фигурица

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Пар производа у ранцу

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

У овом задатку ефикасно решење захтева комбиновање неколико алгоритамских техника. Иако се на први поглед може помислити да се овај задатак решава као класичан проблем ранца, динамичким програмирањем, велика димензија капацитета ранца указује на то да се задатак мора решавати другачије.

Сортирање, два показивача и максимуми префикса

Ако оба низа уредимо тако да масе расту, можемо у једном низу да кренемо од почетка (од најлакшег предмета) а у другом од краја. На тај начин брзо долазимо до парова предмета који су у збиру испод лимита масе, а при томе најближи том лимиту. Ако уједно одржавамо низ максимума цена префикса другог низа, тада лако можемо одредити најскупљи предмет другог који се може узети у комбинацији са тренутним предметом првог низа.

Прикажимо рад овог алгоритма на једном примеру. Нека су цене и тежине мушких поклона дате у левој, а женских у десној колони, при чему смо поклоне сортирали по тежинама и нека је капацитет ранца једнак 20.

```
m1 c1                m2 c2         max_prefiksa_c2
4   3                5   6         6
7   9                9   4         6
12  4                10  12        12
16  11               14  9         12
19  2                18  7         12
```

- Покушавамо да за први мушки поклон (то је (4, 3)) пронађемо скуп поклона који се могу са њим упарити. Пошто је други низ сортиран, то ће бити неки поклони који се налазе на почетку тог низа. Крећемо од краја, елиминишемо последњи женски поклон јер је збир 4 + 18 већи од 20 и проналазимо да је последњи женски поклон који се може упарити са првим мушким онај који има масу 14. Потребно је пронаћи најскупљи женски поклон који има масу мању или једнаку 14. У томе нам помаже последња колона у којој су записани максимуми префикса. За поклон чија је маса 14 можемо очитати вредност 12, што значи да постоји неки женски поклон чија је маса мања или једнака 14 који има вредност 12 (то је поклон (10, 12)). Дакле, ако купимо први мушки поклон, тада је највећа цена коју можемо постићи једнака 3 + 12 = 15.
- Прелазимо на други мушки поклон (то је (7, 9)) и одређујемо женске поклоне са којима се он може комбиновати. Веома важна напомена је то да се поклони који се нису могли укомбиновати са претходним мушким поклонима, не могу укомбиновати ни са текућим (јер је он још тежи од претходних). Дакле, довољно је само међу оним поклонима који су се могли укомбиновати са претходним поклоном наћи оне који се могу укомбиновати са текућим. Пошто је низ женских поклона сортиран по тежини, анализирамо и евентуално елиминишемо елементе са његовог краја (тј. краја оног дела низа где смо се претходно зауставили). Поклон масе 14 се не може комбиновати са поклоном масе 7 (јер им је укупна маса 21 већа од носивости ранца 20). Најтежи женски поклон који се може упарити са оним мушким масе 7 је онај чија је маса 10. Поново на основу низа максимума префикса очитавамо да је највреднији женски поклон чија маса не прелази 10 онај чија је вредност 12 (то је опет (10, 12)), па се његовим комбиновањем са мушким поклоном (7, 9) добија вредност 12 + 9 = 21, што је боље од претходне.

- Прелазимо на следећи мушки поклон (то је (12, 4)), елиминишемо женске поклоне (10, 12) и (9, 4) јер се они не могу комбиновати са мушким поклоном масе 12 и закључујемо да је једини женски поклон који се може комбиновати са (12, 4) поклон (5, 6). Тиме добијамо вредност 4+6 = 10, што је лошије од претходне.
- Преласком на следећи мушки поклон (то је (16, 11)), елиминишемо и женски поклон (5, 6) и закључујемо да се ни један мушки поклон који је тежак 16 или више не може укомбиновати ни са једним женским поклоном.

Дакле, оптимално упаривање је упаривање поклона (7, 9) и (10, 12).

Сложеност сортирања је $O(n \log n)$. Након тога максимуме префикса израчунавамо у сложености $O(n)$ и низ анализирамо техником два показивача у сложености $O(n)$. Укупном сложеношћу, дакле, доминира сортирање и решење је сложености $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// proizvode predstavljamo uredjenim parom (tezina, cena)
typedef pair<int, int> Proizvod;

// ucitavanje niza proizvoda sa standardnog ulaza
vector<Proizvod> ucitajProizvode() {
  int n;
  cin >> n;
  vector<pair<int, int>> proizvodi(n);
  for (int i = 0; i < n; i++) {
    int cena, tezina;
    cin >> cena >> tezina;
    proizvodi[i] = make_pair(cena, tezina);
  }
  return proizvodi;
}

int main() {
  // ucitavamo podatke o proizvodima i maksimalnu tezinu koja moze da
  // stane u ranac
  vector<Proizvod> proizvodi1 = ucitajProizvode();
  vector<Proizvod> proizvodi2 = ucitajProizvode();
  int maksTezina;
  cin >> maksTezina;

  // sortiramo proizvode u oba niza po tezini
  sort(begin(proizvodi1), end(proizvodi1));
  sort(begin(proizvodi2), end(proizvodi2));

  // dimenzije nizova proizvoda
  int n1 = proizvodi1.size(), n2 = proizvodi2.size();

  // za svaku poziciju u sortiranom nizu proizvoda iz druge prodavnice
  // odredjujemo maksimalnu cenu proizvoda striktno pre te pozicije
  vector<int> maksCenaDo2(n2 + 1);
  maksCenaDo2[0] = 0;
  for (int i = 1; i <= n2; i++)
    maksCenaDo2[i] = max(maksCenaDo2[i-1], proizvodi2[i-1].second);

  // maksimalni zbir cena dva proizvoda koja mogu da stanu u ranac
```

```
  int maksCena = 0;
  // tehnika dva pokazivaca: prvi niz obilazimo u rastucem, a drugi u
  // opadajucem redosledu tezina

  // analiziramo svaki proizvod iz prvog niza (koji nije sam za sebe pretezak)
  int i1 = 0, i2 = n2-1;
  while (i1 < n1 && proizvodi1[i1].first < maksTezina) {
    // eliminisemo preteske proizvode iz drugog niza
    while (i2 >= 0 && proizvodi1[i1].first + proizvodi2[i2].first > maksTezina)
      i2--;

    // ako ne postoji proizvod iz drugog niza koji se moze
    // upariti sa tekucim proizvodom iz prvog niza, mozemo prekinuti pretragu
    if (i2 < 0)
      break;

    // azuriramo maksimalnu cenu
    maksCena = max(maksCena, proizvodi1[i1].second + maksCenaDo2[i2+1]);

    // prelazimo na naredni proizvod iz prvog niza
    i1++;
  }
  cout << maksCena << endl;

  return 0;
}
```

Задатак: Број сегмената са различитим елементима

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Груба сила

Једноставно решење добијамо ако анализирамо све сегменте и за сваки проверимо да ли има све различите елементе. Набрајање свих непразних сегмената можемо извршити помоћу двоструке петље (слично као у задатку Све подречи). Проверу да ли су сви елементи низа различити можемо извршити наивно, проверавајући да ли сваки наредни елемент низа припада претходном делу низа.

Сложеност такве провере је $O(n^2)$, па пошто је сегмената укупно $O(n^2)$ ово доводи до огромне укупне сложености $O(n^4)$.

Проверу можемо извршити и ефикасније (слично као, на пример, у задатку Различите цифре), ако елементе низа које смо обишли током провере сместимо у неку ефикасну структуру података за представљање скупа и онда за сваки наредни елемент проверимо да ли том скупу припада (ако припада, у низу има дупликата), а ако не припада, додамо га у скуп (за то је најбоље користити библиотечке колекције `set` или `unordered_set` у језику C++, а ако је домен елемената ограничен, онда и низ логичких променљивих где је `i` постављено на `true` ако и само ако елемент `i` припада скупу).

Ако је сложеност скуповних операција $O(1)$, тада сложеност провере да ли су сви елементи различити постаје сложености $O(n)$, а укупна сложеност пада на $O(n^3)$.

```
#include <iostream>
#include <vector>

using namespace std;

// provera da li su svi elementi segmenta [pocetak, kraj] razliciti
bool razliciti(const vector<int>& a, int pocetak ,int kraj) {
```

```
  for (int i = pocetak; i < kraj; i++)
    for (int j = i + 1; j <= kraj; j++)
      if (a[i] == a[j])
        return false;
  return true;
}

int main() {
  // ucitavamo elemente niza
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ukupan broj nepraznih segmenata niza sa svim razlicitim
  // elementima
  int broj = 0;
  // proveravamo sve segmente
  for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n; j++)
      // provera da li segment a[i], a[i+1], ..., a[j] sadrzi sve
      // razlicite elemente
      if (razliciti(a, i, j))
        broj++;

  // ispisujemo konacan rezultat
  cout << broj << endl;

  return 0;
}
```

Максимални сегменти - два показивача

Посматрајмо низ $1, 2, 9, 3, 4, 9, 5, 6$. Његови сегменти са свим различитим елементима су $(1, 2)$, $(1, 2, 9)$, $(1, 2, 9, 3)$, $(1, 2, 9, 3, 4)$, $(2, 9)$, $(2, 9, 3)$, $(2, 9, 3, 4)$, $(9, 3)$, $(9, 3, 4)$, $(3, 4)$, $(3, 4, 9)$, $(3, 4, 9, 5)$, $(3, 4, 9, 5, 6)$, $(4, 9)$, $(4, 9, 5)$, $(4, 9, 5, 6)$, $(9, 5)$, $(9, 5, 6)$, $(5, 6)$ и (6).

Међу њима се истичу два сегмента: $(1, 2, 9, 3, 4)$ и $(3, 4, 9, 5, 6)$. Они су карактеристични по томе што се не могу проширити ни на лево, ни на десно тако да и даље остану са свим различитим елементима. Дакле, они су на неки начин максимални. Сви други сегменти су подсегменти једног од ова два. При томе, је сегмент $(3, 4)$ подсегмент и једног и другог.

Ово инспирише наредни алгоритам (који је у основи сличан ономе у задатку Број растућих сегмената, али је донекле компликованији јер се за разлику од максималних сегмената у том задатку, у овом задатку суседни максимални сегменти могу преклапати). За дати низ, потребно је идентификовати све максималне подсегменте (оне којима су сви елементи различити и који се не могу проширити ни на лево ни на десно тако да им елементи и даље остану различити). За сваки од њих могуће је одредити број непразних подсегмента – ако је сегмент дужине d, број његових непразних подсегмената је $\frac{d(d-1)}{2}$ (објасни зашто). Сваки сегмент са различитим елементима подсегмент је неког максималног тако да су овим сви тражени сегменти покривени. Међутим, проблем је то што смо неке подсегменте рачунали два пута. Наиме, када се максимални подсегменти преклапају, сви подсегменти дела на којем се они преклапају се рачунају и за леви и за десни максимални сегмент.

Илуструјмо ово још једним примером.

```
1 4 3 2 4 2 3 1 4 3 2 1

1 4 3 2
    3 2 4
```

```
4 2 3 1
  2 3 1 4
      1 4 3 2
        4 3 2 1
```

Приказани су сви максимални подсегменти и њихове дужине су редом 4, 3, 4, 4, 4, 4. Лако је идентификовати и делове на којима се преклапају (прва два се преклапају на $(3, 2)$, други и трећи на (4), наредна два на $(2, 3, 1)$, затим $(1, 4)$ и последња два на $(4, 3, 2)$. Дужине преклопа су редом 2, 1, 3, 2, 3. Тражени број сегмената се може онда добити као: $\frac{4(4-1)}{2} - \frac{2(2-1)}{2} + \frac{3(3-1)}{2} - \frac{1(1-1)}{2} + \frac{4(4-1)}{2} - \frac{3(3-1)}{2} + \frac{4(4-1)}{2} - \frac{2(2-1)}{2} + \frac{4(4-1)}{2} - \frac{3(3-1)}{2} + \frac{4(4-1)}{2}$

тј. $6 - 1 + 3 - 0 + 6 - 3 + 6 - 1 + 6 - 3 + 6 = 25$.

Сада треба разрешити како детектовати максималне подсегменте и како идентификовати пресеке узастопних максималних подсегмената. Користићемо технику два показивача, слично као у задацима Најкраћа подниска која садржи све дате карактере или . Основна идеја је да сегменте проширујемо надесно, све док су елементи различити, а онда када се појави дупликат, да их сужавамо с леве стране, све док се дупликат не уклони.

Претпоставимо да сваки максимални сегмент представљамо са два његова краја тј. да је облика $[p, k)$. Максимални сегмент који почиње на позицији p одређујемо тако што померамо његов десни крај удесно све док се на позицији k не појави елемент који већ припада сегменту $[p, k)$ или се не дође до краја низа. Леви крај првог максималног сегмента је 0. Зато крећемо од празног сегмента $[0, 0)$. Када знамо неки максимални сегмент који није последњи (када је $k < n$), почетак наредног максималног сегмента одређујемо тако што почетак p померамо удесно све док не пређемо елемент који је једнак елементу на позицији k (тај елемент је спречавао да се максимални сегмент прошири иза позиције k). Када p померимо иза тог елемента, тада је сегмент $[p, k)$ тачно пресек два максимална сегмента. Након тога десни крај опет померамо удесно, одређујући тако наредни максимални сегмент.

Илуструјмо рад алгоритма на претходном примеру.

```
0  1  2  3  4  5  6  7  8  9  10 11   - i
1  4  3  2  4  2  3  1  4  3  2  1    - a[i]
```

- Крећемо од сегмента $[0, 0)$.
- Померамо десни крај све док не стигнемо до вредности $k = 4$ и тако налазимо максимални сегмент $[0, 4)$ - даље проширивање није могуће јер се $a_4 = 4$ већ налази у низу на позицији 1 ($a_1 = a_4 = 4$). Дужина сегмента је 4 и бројач увећавамо за $\frac{4(4-1)}{2}$ на 6.
- Након тога померамо леви крај све док се не пређе елемент једнак елементу на позицији $k = 4$, тј. елементу $a_4 = 4$. Пошто је $a_1 = 4$, постављамо p на 2. У том тренутку сегмент $[2, 4)$ представља пресек претходног и наредног максималног сегмента и тада се бројач умањује за $\frac{2(2-1)}{2}$ и постаје 5.
- Након тога померамо опет десну границу на вредност док не стигнемо до сегмента $[2, 5)$ који не може даље да се прошири јер је $a_5 = a_3 = 2$. Бројач се тада увећава за $\frac{3(3-1)}{2}$ и постаје 8.
- Затим се p помера док не прескочи вредност 2 и не стигне до 4, одређује се да је пресек сегмент $[4, 5)$, бројач се умањује за $\frac{1 \cdot (1-1)}{2}$ итд.
- ...
- Поступак се завршава када се пресек $[8, 11)$ прошири до сегмента $[8, 12)$. Бројач се тада увећава за $\frac{4(4-1)}{2}$ и пошто је $k = n$ главна петља се прекида и алгоритам завршава са радом (у том тренутку вредност бројача је 25).

Остаје питање како пре евентуалног проширивања сегмента удесно, тј. повећавања границе k проверити да ли се a_k налази већ у сегменту $[p, k)$. Једно решење је линеарна претрага, међутим, она води ка неефикасном решењу, тако да је пожељно боље решење. Потребно је одржавати скуп елемената који се јављају у сегменту $[p, k)$ и проверити да ли a_k припада том скупу. Ако не припада, тада се приликом повећања вредности k скуп проширује елементом a_k. Приликом одређивања наредног максималног сегмента и померања леве границе p, пре увећавања p потребно је из скупа уклонити елемент a_p. Остаје питање како репрезентовати скуп, да би операције испитивања припадности елемента скупу, додавања елемента у скуп и уклањања елемента из скупа биле ефикасне. Најбољи начин је коришћење библиотечких колекција `set` или `unordered_set` у језику C++. Ако је домен елемената низа ограничен, тада је скуп могуће представити и једноставније асоцијативним низом логичких вредности тако да је на позицији `i` вредност тачно ако и само ако елемент `i` припада скупу.

Број p је могуће увећати највише n пута, исто и као број k. Дакле, укупан број корака је $O(n)$, а у сваком кораку је доминантна операција рад са скупом елемената (провера припадности, додавање или брисање). Ако

претпоставимо да су ове операције сложености $O(1)$ (што је случај ако се скупови имплементирају помоћу хеш-табела), онда је укупна сложеност $O(n)$. Чак и да су скуповне операције сложености $O(\log n)$ (што је случај када се скуп имплементира помоћу бинарног стабла), укупна сложеност је $O(n \log n)$, што је и даље савим прихватљиво.

```
#include <iostream>
#include <vector>
#include <unordered_set>

using namespace std;

// broj podsegmenata (duzine bar 2) segmenta date duzine
int brojPodsegmenata(int duzina) {
  return duzina * (duzina - 1) / 2;
}

int main() {
  // ucitavamo niz elemenata
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ukupan broj segmenata niza ciji su svi elementi razliciti
  int broj = 0;

  // tekuci segment [pocetak, kraj)
  int pocetak = 0, kraj = 0;
  // skup elemenata u tekucem segmentu
  unordered_set<int> s;

  while (true) {
    // prosirujemo segment [pocetak, kraj) bez duplih elemenata sve
    // dok je to moguce
    while (kraj < n && s.find(a[kraj]) == s.end()) {
      s.insert(a[kraj]);
      kraj++;
    }

    // svi neprazni podsegmenti segmenta [pocetak, kraj) su bez duplih
    // elemenata i njihov broj dodajemo na ukupan broj segmenata
    int duzina = kraj - pocetak;
    broj += brojPodsegmenata(duzina);

    // ako segment [pocetak, kraj) nije moguce produziti nadesno
    // zavrsavamo postupak
    if (kraj >= n)
      break;

    // pomeramo pocetak udesno sve dok ne uklonimo element koji je
    // jednak a[kraj], tako da segment [pocetak, kraj] ne sadrzi vise
    // duple elemente
    while (a[pocetak] != a[kraj]) {
      s.erase(a[pocetak]);
      pocetak++;
    }
    s.erase(a[pocetak]);
    pocetak++;
```

```
    // podsegmenti segmenta [pocetak, kraj) ce biti racunati kao
    // neprazni podsegmenti dva maksimalna segmenta (onog u ovoj i
    // onog u narednoj iteraciji), pa zbog toga moramo da ih uklonimo
    // iz brojanja u ovoj iteraciji (oni ce biti ubrojani u narednoj)
    duzina = kraj - pocetak;
    broj -= brojPodsegmenata(duzina);
  }

  // ispisujemo ukupan broj segmenata sa razlicitim elementima
  cout << broj << endl;

  return 0;
}
```

Задатак: Конференција

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Најкраћа подниска која садржи све дате карактере

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Квадратно-троугаони бројеви

Размотримо распоређивање куглица у правилне квадратне и троугаоне облике. На пример, 36 куглица можемо распоредити у квадрат димензије 6×6 (у 6 врста имамо по 6 куглица) и правоугли троугао димензија 8×8 (у врстама имамо редом 1, 2, . . ., 8 куглица).

```
        *
        **
******  ***
******  ****
******  *****
******  ******
******  *******
******  ********
```

Напиши програм који одређује све бројеве куглица мање и једнаке од дате границе који се могу правилно распоредити у неки квадрат и неки троугао.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^{17}$).

Излаз: На стандардни излаз исписати све квадратно-троугаоне бројеве који су мањи или једнаки n.

Пример

Улаз	*Излаз*
2000	1
	36
	1225

Задатак: Кружни пут

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Два показивача

Ефикасан алгоритам је могуће добити ако се примети да се за многе почетне станице претрага не мора вршити из почетка.

Претпоставимо да смо закључили да није могућ кружни пут почев од станице чији је редни број p (почетак), јер смо установили да није могуће направити прелаз до неке станице c (циљ). Нека је c прва таква станица. То значи да је укупно растојање $R_{p,c}$ од станице p до станице c строго веће од укупне количине бензина $B_{p,c-1}$ коју можемо наточити од станице p до станице $c-1$, тј. да је $R_{p,c} > B_{p,c-1}$. Потребно је проверити да ли је тај прелаз могуће направити ако би почетак био у станици $p+1$. То ће бити могуће ако је $R_{p+1,c} \leq B_{p+1,c-1}$, што значи да треба да израчунамо та два збира и упоредимо их. Можемо приметити да је та два збира могуће добити инкрементално, на основу познатих збирова $R_{p,c}$ и $B_{p,c-1}$. Наиме, важи да је $R_{p,c} = r_p + R_{p+1,c}$, где је r_p растојање од станице p до станице $p+1$. Зато је $R_{p+1,c} = R_{p,c} - r_p$. Слично, важи да је $B_{p+1,c-1} = B_{p,c-1} - b_p$, где је b_p количина бензина која се може наточити у станици p. Ако је $R_{p+1,c} < B_{p+1,c-1}$, до станице c не можемо да стигнемо ни ако кренемо из станице $p+1$ и одмах је можемо елиминисати и прећи на наредну станицу $p+2$. Овакво померање почетне станице надесно можемо вршити све док не установимо да смо нашли почетак такав да кренувши од њега можемо да досегнемо станицу c (то може да се догоди и ако вредност p достигне вредност c, тј. ако c постане почетна станица). Дакле, када установимо да не можемо стићи до неке станице c, повећавамо p све док не наиђемо на неку вредност p' за коју важи да је $R_{p',c} \leq B_{p',c-1}$ (ако је $p' = c$, оба броја су нула, па неједнакост важи). Тада треба увећати c и наставити да ли се може стићи до даљих станица.

Поставља се питање да ли можемо да гарантујемо да се из p' заиста може стићи до станице c тј. да ли се можда дешава да се негде раније током пута од p' до c путања прекида. Важи да је $R_{p,c} = R_{p,p'} + R_{p',c}$ и да је $B_{p,c-1} = B_{p,p'-1} + B_{p',c-1}$. Пошто је $R_{p,c} > B_{p,c-1}$, важи да је $R_{p,p'} + R_{p',c} > B_{p,p'-1} + B_{p',c-1}$, тј. да је $R_{p',c} - B_{p',c-1} > B_{p,p'-1} - R_{p,p'}$. Пошто је $R_{p',c} \leq B_{p',c-1}$, важи да је $0 \geq R_{p',c} - B_{p',c-1} > B_{p,p'-1} - R_{p,p'}$, тј. да је $R_{p,p'} > B_{p,p'-1}$. Ако уместо станице c размотримо било коју станицу c' између p' и c, важи да је $R_{p,c'} = R_{p,p'} + R_{p',c'}$ и да је $B_{p,c'-1} = B_{p,p'-1} + B_{p',c'-1}$. Стога је $R_{p',c'} - B_{p',c'-1} = (R_{p,c'} - R_{p,p'}) - (B_{p,c'-1} - B_{p,p'-1}) = (R_{p,c'} - B_{p,c'-1}) + (B_{p,p'-1} - R_{p,p'})$. Међутим, важи да је $R_{p,c'} \leq B_{p,c'-1}$ (јер је c прва станица за коју важи да је $R_{p,c} > B_{p,c-1}$), па је $R_{p,c'} - B_{p,c'-1} \leq 0$. Пошто је и $B_{p,p'-1} - R_{p,p'} < 0$, један сабирак је непозитиван, а други негативан и важи да је $R_{p',c'} - B_{p',c'-1} < 0$. Стога је $R_{p',c'} < B_{p',c'-1}$. Дакле, кренувши из станице p' сигурно се може стићи редом у све станице закључно са станицом c.

Алгоритам зато можемо засновати на техници два показивача. Одржавамо променљиву p која означава тренутни почетак пута и променљиву c која означава тренутни циљ (иницијализујемо их на 0 и 1), променљиву R која чува тренутни збир $R_{p,c}$ и B која чува тренутни збир $B_{p,c-1}$ (њих иницијализујемо на основу података датих у првој станици).

- Ако је $R \leq B$, тада можемо направити наредни корак тј. прећи на наредну станицу, тако што променљиве R и B увећамо за вредности за станицу c и онда увећамо c по модулу n. Тада треба проверити да ли смо обишли цео круг. Пошто је потребно да разликујемо случај када је смо на самом почетку и када смо обишли цео круг (зато не можемо проверавати да ли се тренутно налазимо у почетној станици, јер је то испуњено и у првој и у последњој итерацији), критеријум заустављања можемо формулисати тако да проверавамо да ли у наредном кораку можемо стићи поново до почетка (што је испуњено ако је $c = p$ и ако је $R \leq B$).
- Ако је $R > B$, тада не можемо направити наредни корак и стићи до станице c. Тада умањујемо вредности R и B за вредности придружене станици p и мењамо почетак преласком на наредну почетну станицу (увећавамо p за 1). Ако наредна станица не постоји (ако након увећања добијемо вредност n), тадак кружни пут не постоји.

У сваком кораку петље се увећава или вредност почетка или вредност циља. При том смо сигурни да се вредност променљиве која представља почетак може увећати највише n пута, док се вредност променљиве која представља циљ може увећати највише $2n$ пута, па је укупно време $O(n)$.

```
#include <iostream>
#include <vector>

using namespace std;

struct Stanica {
  int benzin, rastojanje;
};
```

```
int trazi(vector<Stanica> stanice, int n) {
  // krecemo iz stanice 0 i cilj nam je da dodjemo u stanicu 1
  int poc = 0, cilj = 1;
  // ukupno rastojanje od pocetne do ciljne stanice i
  int rastojanje = stanice[poc].rastojanje;
  // ukupna kolicina benzina od pocetne do tekuce stanice
  int benzin = stanice[poc].benzin;
  while (true) {
    if (rastojanje <= benzin) {
      // mozemo stici do sledece stanice, pa prelazimo na nju
      rastojanje += stanice[cilj].rastojanje;
      benzin += stanice[cilj].benzin;
      cilj = (cilj + 1) % n;
      // ako je naredna stanica pocetak i mozemo stici do nje zavrsili
      // smo kruzni put
      if (cilj == poc && rastojanje <= benzin)
        return poc;
    } else {
      // ne mozemo stici do sledece stanice, pa analiziramo naredni pocetak
      rastojanje -= stanice[poc].rastojanje;
      benzin -= stanice[poc].benzin;
      poc = poc + 1;
      // ako naredni pocetak ne postoji, kruzni put nije moguc
      if (poc == n)
        return -1;
    }
  }
}

int main() {
  int n;
  cin >> n;
  vector<Stanica> stanice(n);
  for (int i = 0; i < n; i++)
    cin >> stanice[i].rastojanje >> stanice[i].benzin;
  cout << trazi(stanice, n) << endl;
  return 0;
}
```

Имплементација може бити реализована и помоћу угнежђених петљи, при чему се у спољашњој петљи помера вредност циља, а у унутрашњој вредност почетка (све док се не утврди да је из тог почетка могуће достићи тренутни циљ).

```
#include <iostream>
#include <vector>

using namespace std;

struct Stanica {
  int benzin, rastojanje;
};

int trazi(vector<Stanica> stanice, int n) {
  // krecemo iz stanice 0 i cilj nam je da dodjemo u stanicu 1
  int poc = 0, cilj = 1;
  // ukupno rastojanje od pocetne do ciljne stanice i
  int rastojanje = stanice[poc].rastojanje;
  // ukupna kolicina benzina od pocetne do tekuce stanice
  int benzin = stanice[poc].benzin;
```

```
  // ako je naredni cilj jednak pocetnoj stanici i ako imamo dovoljno
  // benzina da taj korak napravimo, tada smo nasli kruzni put
  while (cilj != poc || rastojanje > benzin) {
    // dok god ne mozemo da napravimo korak do cilja
    // pomeramo pocetak (najvise do cilja)
    while (rastojanje > benzin) {
      benzin -= stanice[poc].benzin;
      rastojanje -= stanice[poc].rastojanje;
      poc = poc + 1;
      if (poc == n)
        return -1;
    }
    // pravimo korak i pomeramo cilj na narednu stanicu
    rastojanje += stanice[cilj].rastojanje;
    benzin += stanice[cilj].benzin;
    cilj = (cilj + 1) % n;
  }
  return poc;
}

int main() {
  int n;
  cin >> n;
  vector<Stanica> stanice(n);
  for (int i = 0; i < n; i++)
    cin >> stanice[i].rastojanje >> stanice[i].benzin;
  cout << trazi(stanice, n) << endl;
  return 0;
}
```

Задатак: Двоструко сортирана претрага

Дата је матрица у којој су све врсте и све колоне сортиране растући. Напиши програм који ефикасно проналази елементе у таквој матрици.

Улаз: Са стандардног улаза уносе се димензије матрице m и n ($1 \leq m, n \leq 1000$), а затим и елементи матрице (елементи сваке врсте у посебном реду, раздвојени размацима). Елементи су цели бројеви између -10^5 и 10^5. Након тога учитава се у сваком реду до краја улаза по један број који се тражи у матрици.

Излаз: За сваки број који се тражи у матрици на стандардни излаз исписати колико пута се појављује у матрици.

Пример

Улаз	*Излаз*
4 5 1 3 5 8 10 4 7 9 11 15 5 9 13 14 20 8 11 14 16 22 11 12 13	2 0 1

Решење

Линеарна претрага

Наиван начин је да сваки елемент тражимо линеарном претрагом, тако што у угнежђеним петљама пролазимо кроз сваки елемент матрице. Разни начини имплементације линеарне претраге приказани су у задатку Негативан број.

Сложеност овог приступа била би $O(m \cdot n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
      if (A[i][j] == x)
        broj++;
  }
  return broj;
}

int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x) << endl;
  return 0;
}
```

Бинарна претрага

Пошто су елементи сваке врсте сортирани, на претрагу сваке врсте могуће је применити поступак бинарне претраге. Тај поступак је детаљно описан, на пример, у задатку Провера бар-кодова.

Ако претрагу вршимо по врстама, онда можемо употребити библиотечку функцију. Ако бисмо претрагу вршили по колонама, тада не бисмо могли да применимо библиотечку функцију.

Сложеност тог приступа је $O(m \cdot \log n)$. Ако има много више врста него колона, ефикасније би било претрагу вршити по колонама (сложеност би тада била $O(n \cdot \log m)$).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// ovo je dovoljno efikasno zbog move semantike c++-11
vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
```

```
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  for (int i = 0; i < m; i++)
    if (binary_search(A[i].begin(), A[i].end(), x))
      broj++;
  return broj;
}

int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x) << endl;

  return 0;
}
```

Оптимизована бинарна претрага

У решењу са бинарном претрагом смо искористили чињеницу да су врсте сортиране, али не и да су колоне сортиране. На основу тога, можемо оптимизовати бинарну претрагу. Уместо претраге за тачним елементом, у сваком кораку можемо вршити претрагу за први елементом који је већи или једнак траженом. Тај поступак је описан, на пример, у задатку Први већи и последњи мањи.

У наредној врсти претрагу можемо вршити само од почетка па до позиције на којој се у претходној врсти налазио елемент такав елемент (јер се у наредној врсти, због сортираности колона, на тој позицији налази елемент који је строго већи од траженог, а због сортираности колона, такви су и сви елементи те врсте иза те позиције).

Прикажимо како да пребројимо појављивања елемента 11 у следећом матрици.

```
1 3 5 8 10
4 7 9 11 15
5 9 13 14 20
8 11 14 16 22
```

- У првој врсти не постоји елемент који је већи или једнак 11, па се у наредном кораку претражује цела врста.
- У другој врсти се елемент 11 налази на позицији 3. Зато се у наредном кораку претражују само прва три елемента.
- У трећој врсти се први елемент који је већи или једнак од 11 налази на позицији 2 (то је елемент 13), па се у последњој врсти претражују само прва два елемента.
- У четвртој врсти се елемент 11 налази на позицији 1.

И даље је у најгорем случају сложеност $O(n \cdot \log m)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cassert>
```

```cpp
using namespace std;

// ovo je dovoljno efikasno zbog move semantike c++-11
vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  for (int i = 0; i < m && n > 0; i++) {
    auto end = next(A[i].begin(), n);
    auto it = lower_bound(A[i].begin(), end, x);
    if (it != end) {
      n = distance(A[i].begin(), it);
      if (*it == x)
        broj++;
    }
  }
  return broj;
}

int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x) << endl;

  return 0;
}
```

Претрага из доњег левог угла

Задатак веома елегантно и ефикасно у линеарној сложености можемо решити техником два показивача.

Посматрајмо део полазне матрице (подматрицу) у чијем се доњем левом углу налази елемент a_{vk} и који се простире до горњег десног угла полазне матрице. Пошто су елементи у врстама и колонама строго растући, сви елементи у колони k изнад елемента a_{vk} су мањи од њега, док су сви елементи у врсти v десно од елемента a_{vk} већи од њега.

На почетку посматрамо целу матрицу (тј. почињемо од елемента a_{vk} за $v = m - 1$ и $n = 0$).

- Ако је елемент a_{vk} већи од траженог, тада се тражени елемент не може налазити у последњој врсти посматране подматрице и претрагу можемо наставити у подматрици којој је доњи леви угао $a_{(v-1)k}$. Ако је нова подматрица празна (што се дешава када је $v = 0$), претрагу није потребно даље настављати. Претрагу нове подматрице настављамо на потпуно исти начин као и полазне (тако што умањимо индекс v за 1).
- Ако је елемент a_{vk} мањи од траженог, тада се тражени елемент не може налазити у првој колони посматране подматрице и претрагу можемо наставити у подматрици којој је доњи леви угао $a_{v(k+1)}$. Ако је нова подматрица празна (што се дешава када је $k = n - 1$), претрагу није потребно даље настављати.

Претрагу нове подматрице настављамо на потпуно исти начин као и полазне (тако што увећамо индекс k за 1).

- Ако је елемент a_{vk} једнак траженом, тада можемо увећати бројач појављивања траженог елемента. Пошто су елементи у врстама и колонама строго растући, знамо да се елемент не може налазити ни изнад ни десно од елемента a_{vk}, тако да наставак претраге можемо наставити у подматрици којој је доњи десни угао $a_{(v-1)(k+1)}$. Ако је нова подматрица празна (што се дешава када је $k = n-1$), претрагу није потребно даље настављати. Претрагу нове подматрице настављамо на потпуно исти начин као и полазне (тако што увећамо индекс k за 1).

Прикажимо како да пребројимо појављивања елемента 11 у следећом матрици.

```
1 3 5 8 10
4 7 9 11 15
5 9 13 14 20
8 11 14 16 22
```

- Крећемо из доњег левог угла и поредимо 8 и 11. Пошто је 8 мање од 11, мањи од 11 су и сви елементи прве врсте (јер су врсте и колоне сортиране) и целу прву колону можемо надаље занемарити. Зато се померамо удесно.
- Наишли смо на елемент 11, па увећавамо његов број појављивања. Пошто су врсте и колоне сортиране строго растуће, знамо да се ни изнад тог елемента 11, ни десно од њега не могу налазити нови елементи 11, па другу колону и последњу, четврту врсту можемо надаље занемарити. Зато се померамо горе-десно.
- Поредимо елемент 13 и 11. Пошто је 13 веће од 11, знамо да се десно од тог елемента 13 не могу налазити елементи 11, па целу претпоследњу, трећу врсту можемо надаље занемарити. Померамо се навише.
- Поредимо елемент 9 и 11. Пошто је 9 мање од 11, знамо да се изнад елемента 9 не могу налазити елементи 11, па целу трећу колону можемо надаље занемарити. Померамо се надесно.
- Наишли смо још један елемент 11. Знамо да се ни изнад ни десно од њега не могу налазити нови елементи 11. Зато елиминишемо целу четврту колону и другу врсту.
- Поредимо једини преостали елемент 10. Он је мањи од 11, па се померамо надесно. Пошто тиме излазимо из опсега матрице, претрага се завршава.

Сложеност овог приступа у најгорем случају је $O(m+n)$. Напоменимо да је ово решење лошије од бинарне претраге када имамо мали број прилично дугачких врста.

```
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  int v = m - 1, k = 0;
  while (v >= 0 && k < n)
```

```
    if (A[v][k] < x)
      k++;
    else if (A[v][k] > x)
      v--;
    else {
      broj++;
      v--; k++;
    }
  return broj;
}

int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x) << endl;
  return 0;
}
```

Претходни поступак сасвим природно може бити описан и рекурзивном функцијом.

```
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x, int v, int k) {
  if (v < 0 || k >= (int)A[0].size())
    return 0;
  if (A[v][k] < x)
    return brojPojavljivanja(A, x, v, k+1);
  else if (A[v][k] > x)
    return brojPojavljivanja(A, x, v-1, k);
  else
    return 1 + brojPojavljivanja(A, x, v-1, k+1);
}

int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int m = A.size();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x, m-1, 0) << endl;
  return 0;
}
```

Глава 3

Конструкција алгоритама рекурзијом тј. индукцијом

Један од основних механизама конструкције алгоритама подразумева да се до решења проблема долази тако што се проблем сведе на решавање једног или више потпроблема истог облика, али мање димензије. Свођење, наравно, не може да тече у недоглед, већ је потребно да проблеме мале димензије умемо да решимо директно, без даљег свођења. На пример, проблем димензије 0 се решава директно, док се за свако $n > 0$, проблем димензије n своди на проблем димензије $n - 1$.

Имплементација овог поступка може бити реализована на два начина.

- Могуће је дефинисати рекурзивну функцију (функције која позива саму себе), којој се преко улазних параметара (уз евентуално коришћење додатних, глобалних променљивих) прослеђује опис проблема који се тренутно решава. Унутар функције се врши анализа да ли је прослеђени проблем довољно мале димензије да би се могао директно решити или се његово решење добија тако што функција позове сама себе да реши један или више мањих потпроблема.

- Могуће је дефинисати итеративни поступак, који подразумева да се у петљи променљиве ажурирају, кренувши од решења проблема мале димензије па проширујући решење мало по мало, све док се не дође до решења проблема тражене димензије.

Без обзира на то да ли се користи рекурзивна или итеративна имплементација, у основни оваквих алгоритама лежи исти поступак и њихова коректност следи на основу приниципа математичке индукције. Случај који се директно решава представља базу индукције. Индуктивна је претпоставка да су потпроблеми коректно решени и на основу те индуктивне претпоставке се доказује да се полазни проблем коректно решава. Стога ћемо често говорити о **индуктивно-рекурзивној** конструкцији и често ћемо приказивати и рекурзивну и нерекурзивну имплементацију (до које ћемо долазити било директно, било ослобађањем рекурзије).

3.1 Елементарне петље изражене рекурзивно

Задатак: Бројеви од a до b

Ако су дати цели бројеви a и b, написати програм који исписује редом све целе бројеве у задатом интервалу $[a, b]$.

Улаз: У првој линији стандардног улаза налази се цео број a, а у другој је цео број b ($-1000 \leq a \leq 1000, -1000 \leq b \leq 1000$).

Излаз: На стандардном излазу исписују се редом сви цели бројеви из интервала, у свакој линији по један број.

Пример

Улаз	*Излаз*
3	3
6	4
	5
	6

Решење

Задатак можемо решити и рекурзивном функцијом (мада је то решење неефикасније и неприродније од оног итеративног). Ако је $a > b$, тада је низ бројева празан и не треба исписивати ништа. У супротном можемо исписати број a, а затим све бројеве из интервала $[a + 1, b]$.

```
#include <iostream>
using namespace std;

void brojevi_od_a_do_b(int a, int b) {
  if (a <= b) {
    cout << a << endl;
    brojevi_od_a_do_b(a+1, b);
  }
}

int main() {
  int a, b;
  cin >> a >> b;
  brojevi_od_a_do_b(a, b);
  return 0;
}
```

Рекурзивно решење може бити организовано и тако да када је $a \leq b$, да се прво испишу бројеви из интервала $[a, b - 1]$, а затим и број b.

```
#include <iostream>
using namespace std;

void brojevi_od_a_do_b(int a, int b) {
  if (a <= b) {
    brojevi_od_a_do_b(a, b-1);
    cout << b << endl;
  }
}

int main() {
  int a, b;
  cin >> a >> b;
  brojevi_od_a_do_b(a, b);
  return 0;
}
```

Задатак: Бројање у игри жмурке

У игри жмурке деца обично броје по пет $(5, 10, 15, 20, \ldots)$. Напиши програм који исписује баш те бројеве.

Улаз: Са стандардног улаза уноси се број x $(100 \leq x \leq 1000)$ дељив са 5.

Излаз: На стандардни излаз исписати бројеве дељиве са 5, почевши од 5 и завршивши са x. Сваки број исписати у посебном реду.

Пример

Улаз	*Излаз*
30	5
	10
	15
	20
	25
	30

Решење

Задатак можемо решити и рекурзивно (мада је то овде неефикасније и неприродније решење).

```
#include <iostream>
using namespace std;

void zmurke(int a, int b) {
  if (a <= b) {
    cout << a << endl;
    zmurke(a+5, b);
  }
}

int main() {
  int x;
  cin >> x;
  zmurke(5, x);
  return 0;
}
```

Задатак: Троцифрени парни бројеви

За дате целе бројеве a и b, написати програм који исписује редом све парне троцифрене бројеве који припадају датом интервалу $[a, b]$

Улаз: Са стандардног улаза учитавају се бројеви a и b ($0 \leq a \leq 1500$ и $a \leq b \leq 1500$).

Излаз: На стандардном излазу исписују се редом (од најмањег до највећег) сви парни троцифрени бројеви, у свакој линији по један број.

Пример

Улаз	*Излаз*
85	100
109	102
	104
	106
	108

Задатак: Одбројавање уназад

Написати програм који одбројава уназад од датог броја до нуле.

Улаз: Са стандардног улаза уноси се природан број a мањи од 100 од којег почиње одбројавање.

Излаз: На стандардном излазу исписују се редом (од највећег до најмањег) сви бројеви од a до нуле. Сваки број приказати у посебној линији.

Пример

Улаз	*Излаз*
3	3
	2
	1
	0

Решење

Задатак можемо решити и рекурзијом (мада је то овде неефикасније и неприродније решење). Ако је a ненегативан број, тада бројеве из интервала $[0, a]$ исписујемо тако што прво испишемо број a, а затим уназад испишемо бројеве из интервала $[0, a-1]$.

```
#include <iostream>

using namespace std;

void ispisi_unazad(int a) {
  if (a >= 0) {
    cout << a << endl;
    ispisi_unazad(a-1);
  }
}

int main() {
  int a;
  cin >> a;
  ispisi_unazad(a);
  return 0;
}
```

Задатак: Подела интервала на једнаке делове

Написати програм којим се исписују вредности n равномерно размакнутих реалних бројева из интервала $[a, b]$, тако да је прва вредност a, а последња b.

Улаз: Прва линија стандардног улаза садржи природан број n $(1 < n \leq 20)$, друга линија садржи реалан број a, а трећа линија реалан број b, при чему је $a < b$.

Излаз: На стандардном излазу приказати редом тражене бројеве, заокружене на пет децимала, сваки у посебној линији.

Пример

Улаз	*Излаз*
5	-1.00000
-1	-0.50000
1	0.00000
	0.50000
	1.00000

Решење

Задатак можемо решити и рекурзивно (мада је то у овој ситуацији неефикасније и неприродније решење). Можемо дефинисати функцију која исписује n бројева на међусобном растојању dx, кренувши од броја a. Ако је број n строго позитиван, исписујемо број a, а затим исписујемо $n-1$ бројева на растојању dx, кренувши од броја $a + dx$.

```
#include <iostream>
#include <iomanip>
using namespace std;

// ispisuje n ravnomerno rasporedjenih realnih brojeva na
// medjusobnom rastojanju dx, pocevsi od broja a
void ravnomerno_rasporedjeni_brojevi(int n, double a, double dx) {
  if (n > 0) {
    cout << setprecision(5) << showpoint << fixed
         << a << endl;
    ravnomerno_rasporedjeni_brojevi(n-1, a+dx, dx);
  }
}
```

```
int main() {
   int n;
   cin >> n;
   double a, b;
   cin >> a >> b;
   double dx = (b - a) / (n - 1);
   ravnomerno_rasporedjeni_brojevi(n, a, dx);
   return 0;
}
```

Задатак: Геометријска серија

Написати програм који за дате природне бројеве a, b исписује бројеве из интервала $[a, b]$, од којих је први број који се исписује једнак a, а сваки следећи је три пута већи од претходног. На пример, за $[a, b] = [5, 50]$ треба исписати 5, 15, 45.

Улаз: Са стандардног улаза се учитавају природни бројеви a ($1 \leq a \leq 50$) и b ($a \leq b \leq 10000$) сваки у посебном реду.

Излаз: На стандардном излазу исписују се сви тражени бројеви, редом (од најмањег до највећег). Сваки број исписати у посебној линији.

Пример

Улаз	*Излаз*
5	5
50	15
	45

Задатак: Збир n бројева

Написати програм којим се одређује збир n датих целих бројева.

Улаз: У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 1000$). У свакој од наредних n линија налази се по један цео број x_i.

Излаз: У првој линији стандарног излаза приказати збир унетих n целих бројева $x_1, \ldots, x_n$.

Пример

Улаз	*Излаз*
4	13
10	
-3	
2	
4	

Задатак: Читање до нуле

Уносе се цели бројеви док се не унесе нула. Написати програм којим се приказује колико је унето бројева, не рачунајући нулу.

Улаз: Свака линија стандардног улаза, изузев последње, садржи цео број различит од нуле. Последња линија садржи нулу.

Излаз: На стандардном излазу у првој линији приказати колко је учитано бројева, не рачунајући нулу.

Пример 1

Улаз	*Излаз*
5	3
-675	
123	
0	

Пример 2

Улаз	*Излаз*
0	0

Задатак: Просек свих бројева до краја улаза

Са стандардног улаза се учитава број поена такмичара на такмичењу из програмирања. Напиши програм који израчунава просечан број поена свих такмичара.

Улаз: Сваки ред стандардног улаза садржи један цео број између 0 и 100. НАПОМЕНА: приликом интерактивног тестирања програма, крај стандардног улаза се означава комбинацијом тастера `ctrl + z` ако се користи оперативни систем Windows тј. `ctrl + d` ако се користи оперативни систем Linux.

Излаз: На стандардни излаз исписати просек заокружен на 5 децимала.

Пример

Улаз	*Излаз*
1	2.50000
2	
3	
4	

Задатак: Прерачунавање миља у километре

Миља је енглеска историјска мера за дужину која износи $1609.344\,\mathrm{m}$. Напиши програм који исписује таблицу прерачунавања миља у километре.

Улаз: Са стандардног улаза се уносе цели бројеви a ($1 \leq a \leq 10$), b ($10 \leq b \leq 100$) и k ($1 \leq k \leq 10$).

Излаз: На стандардни излаз исписати табелу конверзије миља у километре за сваки број миља из интервала $[a, b]$, са кораком k. Број километара заокружити на 6 децимала, а табелу приказати у формату идентичном као у примеру.

Пример

Улаз	*Излаз*
10	10 mi = 16.093440 km
20	12 mi = 19.312128 km
2	14 mi = 22.530816 km
	16 mi = 25.749504 km
	18 mi = 28.968192 km
	20 mi = 32.186880 km

Задатак: Табелирање функције

Аутомобил се креће равномерно убрзано са почетном брзином v_0 (израженом у $\frac{\mathrm{m}}{\mathrm{s}}$) и убрзањем a (израженим у $\frac{\mathrm{m}}{\mathrm{s}^2}$). Укупно време до постизања максималне брзине је T секунди. На сваких Δt секунди од почетка потребно је израчунати пређени пут аутомобила. Напомена: за равномерно убрзано кретање пређени пут након протеклог времена t изражава се са $s = v_0 \cdot t + \frac{a \cdot t^2}{2}$.

Улаз: Са стандардног улаза учитавају се 4 реална броја (сваки је у посебном реду):

- v_0 ($0 \leq v_0 \leq 5$) - почетна брзина
- a ($1 \leq a \leq 3$) - убрзање
- T ($5 \leq T \leq 10$) - укупно време
- Δt ($0.1 \leq \Delta t \leq 2.5$) - интервал

Излаз: На стадардни излаз исписати серију бројева који представљају пређени пут у задатим тренуцима.

Пример

Улаз	*Излаз*
1	0.00000
1	0.62500
2	1.50000
0.5	2.62500
	4.00000

Задатак: Факторијел

Дате су цифре $1, 2, \ldots, n$. Напиши програм који израчунава колико се различитих n-тоцифрених бројева састављених од свих тих цифара може направити (на пример, од цифара $1, 2, 3$ могу се направити бројеви 123, 132, 213, 231, 312 и 321).

Напомена: Број пермутација скупа од n елемената једнак је факторијелу броја n тј. броју $n! = 1 \cdot 2 \cdot \ldots \cdot n$. Размисли зашто је баш тако.

Улаз: Прва линија стандарног улаза садржи природан број n ($1 \leq n \leq 9$).

Излаз: У првој линији стандарног излаза приказати број различитих бројева који се могу направити од цифара $1, 2, \ldots, n$.

Пример 1

Улаз	*Излаз*
5	120

Пример 2

Улаз	*Излаз*
9	362880

Задатак: Степен

Напиши програм који израчунава степен x^n. Покушај да програм напишеш без употребе библиотечких функција и оператора за степеновање.

Улаз: Са стандардног улаза се уноси реалан број x ($0.8 \leq x \leq 1.2$) и цео број n ($0 \leq n \leq 20$).

Излаз: На стандардни излаз испиши вредност x^n заокружену на пет децимала.

Пример

Улаз	*Излаз*
1.1 5	1.61051

Задатак: Једнакост растојања

Становници једне дугачке улице желе да одреде положај на којем ће бити направљена антена за мобилну телефонију. Пошто желе да локацију одреде на најправеднији могући начин, договорили су се да антену сагради на месту на ком ће збир растојања свих оних који се налазе лево од антене до ње, бити једнак збиру растојања свих оних који се налазе десно од антене до ње. Ако су познате координате свих кућа у улици (можемо замислити да су то координате тачака на једној правој), напиши програм који одређује положај антенте.

Улаз: Са стандардног улаза у првој линији се уноси природан број n ($1 \leq n \leq 100$) који представља број станара, а у наредних n линија реални бројеви (од -1000 до 1000) који представљају координате станара (x координате тачака на оси).

Излаз: На стандардни излаз исписати један реалан број који представља тражени положај антене (допуштена је толеранција грешке 10^{-5}).

Пример

Улаз	*Излаз*
5 -7.34 15.6 3.67 -22.17 6.2	-0.80800

Задатак: Средине

Аутомобил путује мењајући брзину током путовања. Познато је да се један део пута кретао равномерно брзином v_1 $\frac{\text{km}}{\text{h}}$, затим се један део пута кретао равномерно брзином v_2 $\frac{\text{km}}{\text{h}}$, и тако даље, све до последњег дела пута где се кретао равномерно брзином од v_n $\frac{\text{km}}{\text{h}}$. Написати програм који одређује просечну брзину аутомобила на том путу и то:

- ако се претпостави да је сваки део пута трајао исто време,
- ако се претпостави да је на сваком делу пута аутомобил прешао исто растојање.

Улаз: Са стандардног улаза уноси се n ($2 \leq n \leq 10$) позитивних реалних бројева: v_1, v_2, …, v_n (za svako v_i важи $30 \leq v_i \leq 120$), након чега следи крај улаза.

Излаз: У првој линији стандардног излаза исписати реалан број заокружен на 2 децимале који представља просечну брзину под претпоставком да је сваки део пута трајао исто време, а у другом реду реалан број заокружен на 2 децимале који представља просечну брзину под претпоставком да је на сваком делу пута аутомобил прешао исто растојање.

Пример

Улаз	*Излаз*
60	50.00
40	48.00

Задатак: Најнижа температура

Дате су температуре у неколико градова. Написати програм којим се одређује, колика је температура у најхладнијем граду.

Улаз: Прва линија стандардног улаза садржи природан број n ($3 \leq n \leq 50$) који представља број градова, а у свакој од наредних n линија налази се цео број t ($-20 \leq t \leq 40$) који представља температуру у одговарајућем граду.

Излаз: На стандардном излазу, у једној линији, приказати температуру у најхладнијем граду.

Пример

Улаз	*Излаз*
5	-13
10	
-12	
22	
-13	
15	

Задатак: Други на ранг листи

На основу резултата такмичењу на којем је учествовало N ученика формирана је ранг листа. Ранг листа се формира у нерастућем поретку по резултатима, од најбољег до најлошијег резултата. Написати програм којим се одређује број поена такмичара који је други на ранг листи.

Улаз: Прва линија стандардног улаза садржи природан број N ($1 \leq N \leq 50000$) који представља број такмичара. У свакој од наредних N линија налази се цео број из интервала $[0, 50000]$, ти бројеви представљају резултате такмичара, који нису сортирани по поенима, већ по бројевима рачунара за којима су такмичари седели.

Излаз: У једној линији стандардног излаза приказати број поена другог такмичара на ранг листи.

Пример

Улаз	*Излаз*
5	95
80	
95	
75	
50	
95	

Решење

Задатак се може решити и дефинисањем рекурзивне функције која одређује највећа два елемента у низу. Ако је низ празан, резултат треба поставити на $(-\infty, -\infty)$, јер је $-\infty$ неутрални елемент за операцију максимума. Пошто су у задатку сви бројеви позитивни, уместо вредности $-\infty$ коју обично не подржавају програмски језици, можемо употребити $(-1, -1)$. Ако низ није празан, рекурзивно проналазимо први и други максимум првих $n-1$ елемената низа и затим учитавамо последњи елемент. Ако је тај последњи елемент већи од првог максимума префикса, тада је он први максимум целог низа, а први максимум префикса је други максимум целог низа. У супротном, ако је последњи елемент низа већи од другог максимума префикса, тада је први

максимум префикса уједно и први максимум целог низа, док је други максимум целог низа последњи елемент x. У супротном, први и други максимум целог низа се поклапају са првим и другим максимумом префикса.

```
#include <iostream>

using namespace std;

// ucitava n elemenata i odredjuje i vraca najveca dva
void dvaNajveca(int n, int& prviMaks, int& drugiMaks) {
  if (n == 0) {
    prviMaks = drugiMaks = -1;
    return;
  }
  dvaNajveca(n-1, prviMaks, drugiMaks);
  int x;
  cin >> x;
  if (x > prviMaks) {
    drugiMaks = prviMaks;
    prviMaks = x;
  } else if (x > drugiMaks)
    drugiMaks = x;
}


int main() {
  ios_base::sync_with_stdio(false);

  // broj elemenata serije
  int n;
  cin >> n;

  int prviMaks, drugiMaks;
  dvaNajveca(n, prviMaks, drugiMaks);

  // ispisujemo vrednost drugog maksimuma
  cout << drugiMaks << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Разлика сума до мах и од мах

Уноси се масе предмета, одредити разлику суме маса предмета до првог појављивања предмета највеће масе и суме маса предмета после првог појављивања предмета највеће масе (предмет највеће масе није укључен ни у једну суму).

Улаз: У првој линији стандардног улаза налази се број предмета n ($1 \leq n \leq 50000$). Свака од наредних n линија садржи по један природан број из интервала $[1, 50000]$, ти бројеви представљају масе сваког од n предмета.

Излаз: У првој линији стандарног излаза приказати тражену разлику маса.

Пример

Улаз	*Излаз*
5	-14
10	
13	
7	
13	
4	

Објашњење

Предмет највеће масе је 13. Збир маса пре његовог првог појављивања је 10, а после његовог првог појављивања је 24. Зато је тражена разлика -14.

Решење

Задатак је могуће решити репно-рекурзивном функцијом. Функција прима потребне статистике о делу низа обрађеном пре њеног позива (максимум, збир елемената пре максимума и збир елемената после максимума) као и преостали број елемената које треба учитати и обрадити. Ако је тај број једнак нули, тада је обрађен цео низ и статистике за обрађени део низа су заправо тражене статистике за цео низ. У супротном, учитавамо нови елемент и вршимо рекурзивни позив у коме се број преосталих елемената умањује за 1 и у коме се ажурирају статистике обрађеног дела низа. Ако учитани број није већи од максимума, тада се мења само збир елемената после максимума тако што се увећа за тај учитани број. У супротном смо пронашли нови максимум, збир елемената пре њега је збир свих претходно обрађених елемената (а њега можемо израчунати сабирањем ранијег збира пре максимума, збира после максимума и максимума), док је збир елемената после максимума 0. У иницијалном рекурзивном позиву можемо учитати само један елемент низа, максимум поставити на његову вредност, а збирове пре и после на нулу.

Ако користимо нотацију $z(n, pre, maks, posle)$, и ако учитавамо редом елементе 8, 7, 10, 9, 6 тада ће се вршити следећи низ рекурзивних позива $z(5) = z(4, 0, 8, 0) = z(3, 0, 8, 7) = z(2, 15, 10, 0) = z(1, 15, 10, 9) = z(0, 15, 10, 15) = (15, 15)$.

```
#include <iostream>
#include <tuple>

using namespace std;

// izracunavanje pod pretpostavkom da je u dosadasnjem delu niza
// najveci element maks, da je zbir elemenata pre njega u tom delu
// niza jednak zbirPre, a zbir elemenata posle njega u tom delu
// niza jednak zbirPosle, ucitava jos n preostalih elemenata niza
// i vraca zbir pre i posle maksimalnog elementa u celom nizu
pair<int, int> zbirPreIPosleMaks(int n, int zbirPre, int zbirPosle,
                                 int maks) {
  if (n == 0)
    return make_pair(zbirPre, zbirPosle);
  // ucitavanje mase narednog predmeta
  int m;
  cin >> m;
  if (m <= maks)
    return zbirPreIPosleMaks(n-1, zbirPre, zbirPosle + m, maks);
  else
    return zbirPreIPosleMaks(n-1, zbirPre + maks + zbirPosle, 0, m);
}

pair<int, int> zbirPreIPosleMaks(int n) {
  // ucitavanje mase prvog predmeta
  int m;
  cin >> m;
  // ucitavanje mase ostalih n-1 predmeta i izracunavanje zbirova
  return zbirPreIPosleMaks(n-1, 0, 0, m);
}
```

```
int main() {
  // broj elemenata niza
  int n;
  cin >> n;
  // ucitavamo niz i odredjujemo zbir pre i posle maksimlanog elementa
  int zbirPre, zbirPosle;
  tie(zbirPre, zbirPosle) = zbirPreIPosleMaks(n);
  // prikaz rezultata
  cout << zbirPre - zbirPosle << endl;

  return 0;
}
```

Задатак: Негативан број

Написати програм којим се проверава да ли међу учитаним бројевима постоји негативан број.

Улаз: Прва линија стандарног улаза садржи природан број n ($1 \leq n \leq 100$) који представља број бројева. Свака од наредних n линија садржи по један реалан број.

Излаз: На стандарном излазу у једној линији исписти `da` ако међу учитаним бројевима постоји негативан број, иначе исписати `ne`.

Пример 1

Улаз	*Излаз*
5 10.89 12.349 -5.9 2.3 -2.45	da

Пример 2

Улаз	*Излаз*
4 100.89 12.349 0 2.3	ne

Задатак: Прва негативна температура

Познате су температуре за сваки дан неког периода. Написати програм којим се одређује редни број дана у том периоду када је температура први пут била негативна.

Улаз: Прва линија стандарног улаза садржи природан број n ($3 \leq n \leq 365$) који представља број дана периода. Свака од наредних n линија садржи по један цео број из интервала $[-5, 40]$, бројеви представљају температуре редом за n дана периода.

Излаз: У првој линији стандарног излаза приказати редни број дана периода када је температура први пут била негативна (дани се броје од 1 до n), ако такав дан не постоји приказати `-1`.

Пример 1

Улаз	*Излаз*
5 12 10 -2 -3 4	3

Пример 2

Улаз	*Излаз*
3 12 10 14	-1

Задатак: Последња негативна температура

Познате су температуре за сваки дан неког периода. Написати програм којим се одређује редни број дана у том периоду када је температура последњи пут била негативна (дани се броје од 1 до n).

Улаз: Прва линија стандардног улаза садржи природан број n ($3 \leq n \leq 365$) који представља број дана периода. Свака од наредних n линија садржи по један цео број из интервала $[-10, 40]$, бројеви представљају температуре редом за n дана периода.

Излаз: У првој линији стандардног излаза приказати редни број дана периода када је температура последњи пут била негативна, ако такав дан не постоји приказати `-1`.

Пример 1

Улаз	*Излаз*
5 2 -8 -2 -3 4	4

Пример 2

Улаз	*Излаз*
3 12 10 14	-1

Решење

Задатак можемо решити и рекурзивном функцијом која учитава n бројева и одређује позицију последњег негативног међу њима. Ако је $n = 0$, нема бројева, па самим тим ни негативних, па функција може да врати -1, као ознаку за то. У супротном, функција рекурзивно одређује позицију последњег негативног међу првих $n - 1$ учитаних бројева, а затим учитава и последњи, n-ти број. Ако је он негативан, то је уједно и последњи негативан, па враћамо његову позицију n (јер позиције бројимо од 1), а у супротном враћамо резултат рекурзивног позива.

```
#include <iostream>

using namespace std;

// ucitava n elemenata i vraca poziciju poslednjeg negativnog
// (pozicije se broje od 1) ili -1 ako takav element ne postoji
int poslednjiNegativan(int n) {
  if (n == 0)
    return -1;
  int p = poslednjiNegativan(n-1);
  int x; cin >> x;
  return x < 0 ? n : p;
}

int main() {
  int n;         //  broj dana koji se analiziraju
  cin >> n;

  cout << poslednjiNegativan(n) << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Бројеви дељиви са 3

Напиши програм који међу унетим бројевима одређује и исписује оне који су дељиви са 3.

Улаз: Са стандардног улаза се најпре учитава природан број n ($1 \leq n \leq 1000$), а потом и n природних бројева из интервала $[1, 10^8]$, сваки у посебном реду.

Излаз: На стандардни излаз исписати све учитане бројеве који су дељиви са 3 (у истом редоследу у ком су учитани). Сваки број исписати у посебној линији.

Пример

Улаз	*Излаз*
5 100 11 12 18 102	12 18 102

Задатак: Просек одличних

Дате су просечне оцене n ученика једног одељења. Написати програм којим се одређује просек просечних оцена свих одличних ученика тог одељења.

Улаз: Прва линија стандарног улаза садржи природан број n ($1 \leq n \leq 100$) који представља број ученика. У наредних n линија налази се по један реалан број из интервала $[2, 5]$. Ти бројеви представљају просеке ученика.

Излаз: У првој линији стандарног излаза приказати просек просечних оцена одличних ученика одељења заокружен на две децимале. Ако одличних ученика нема приказати `-`.

Пример 1

Улаз	*Излаз*
4 3.5 4.75 3 4.5	4.62

Пример 2

Улаз	*Излаз*
4 3.5 3.75 2.80 4.35	-

Задатак: Број и збир цифара броја

Написати програм којим се одређује број и збир цифара декадног записа природног броја.

Улаз: Са стандардног улаза се учитава цео број од 0 до 1000000000.

Излаз: На стандардни излаз се исписују број цифара и збир цифара учитаног броја.

Пример 1

Улаз	*Излаз*
23645	5 20

Пример 2

Улаз	*Излаз*
0	1 0

Решење

Број и збир цифара могуће је одредити и рекурзивном функцијом.

```
#include <iostream>
using namespace std;

int brojCifara(int n) {
  if (n < 10)
    return 1;
  else
    return brojCifara(n / 10) + 1;
}

int zbirCifara(int n) {
  if (n < 10)
    return n;
  else
    return zbirCifara(n / 10) + n % 10;
}

int main() {
  int n;
```

```
  cin >> n;
  cout << brojCifara(n) << " "
       << zbirCifara(n) << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Провера бар-кодова

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Рекурзивна имплементација

Алгоритам бинарне претраге вредности у низу одговара оном који се може применити у игри погађања непознатог броја и који је заснован на половљењу интервала.

Претпоставимо да желимо проверити да ли се елемент x налази у низу a између позиција l и d тј. да ли се налази у затвореном интервалу позиција $[l, d]$. Пронађимо средину овог интервала s.

Ако је $x < a[s]$, пошто је низ сортиран неопадајуће, важи да је x мањи и од свих елемената који су десно од s. Зато претрагу можемо наставити у интервалу $[l, s-1]$. Слично, ако је $x > a[s]$, пошто је низ сортиран неопадајуће, важи да је x веће од свих елемената лево од s тако да претрагу можемо наставити само у интервалу $[s+1, d]$. На крају, ако је $x = a[s]$ тада смо елемент пронашли у низу и то на позицији s. Претрага се може прекинути и када се интервал који се претражује испразни (што ће се десити ако је $l > d$).

Бинарну претрагу можемо имплементирати рекурзивно, дефинисањем функције која проверава да ли се елемент x налази у низу a у интервалу позиција $[l, d]$. Ако је интервал празан, излази се из рекурзије тако што се јавља да елемент није присутан у низу. У супротном се средишњи елемент интервала пореди са траженим елементом x и у зависности од тога да ли је мањи, већи или једнак од x, рекурзивно се тражи у левој половини, десној половини или се пријављује да је пронађен.

Пошто се у сваком кораку интервал $[l, d]$ полови, пошто иницијално крећемо од интервала $[0, n-1]$ и пошто се претрага завршава када се интервал испразни, сложеност претраге једног елемента је $O(\log n)$ (па се m елемената независно претражује у $O(m \log n)$ корака). Нагласимо и да је то разлог што се веома брзо дође до излаза из рекурзије, па у овом конкретном примеру рекурзивна имплементација не заостаје пуно у односу на итеративну. Рекурзија је репна и веома је једноставно ослободити је се.

```
#include <iostream>

using namespace std;

// funkcija proverava da li se u intervalu [l, d] nalazi element x
bool sadrzi(int a[], int l, int d, int x) {
  // prazan interval ne sadrzi element x
  if (l > d)
    return false;

  // nalazimo sredinu intervala
  int s = l + (d - l) / 2;

  // ako je x manji od srednjeg on se moze nalaziti samo u intevalu
  // [a, s-1] (jer je niz sortiran)
  if (x < a[s])
    return sadrzi(a, l, s - 1, x);
  // ako je x veci od srednjeg on se moze nalaziti samo u intevalu
  // [s+1, d] (jer je niz sortiran)
  else if (x > a[s])
    return sadrzi(a, s + 1, d, x);
  else
```

```
    // nasli smo element x na poziciji s
    return true;
}

// funkcija proverava da li se u datom sortiranom nizu a duzine n
// nalazi element x
bool sadrzi(int a[], int n, int x) {
  return sadrzi(a, 0, n-1, x);
}


// maksimalni broj elemenata niza dat tekstom zadatka
const int MAX = 50000;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  int a[MAX];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // broj onih koji postoje u nizu
  int broj = 0;
  // ucitavamo broj po broj do kraja ulaza
  int x;
  while (cin >> x) {
    // ako je broj sadrzan u nizu, uvecavamo brojac
    if (sadrzi(a, n, x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;
  return 0;
}
```

Задатак: Обједињавање

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Рекурзивна имплементација може изгледати овако.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void objedini(const vector<int>& a, int i,
              const vector<int>& b, int j,
              vector<int>& c, int k) {
  if (i < a.size() && j < b.size()) {
    if (a[i] < b[j]) {
      c[k] = a[i];
      objedini(a, i+1, b, j, c, k+1);
```

```
    } else {
      c[k] = b[j];
      objedini(a, i, b, j+1, c, k+1);
    }
  } else if (i < a.size()) {
    c[k] = a[i];
    objedini(a, i+1, b, j, c, k+1);
  } else if (j < b.size()) {
    c[k] = b[j];
    objedini(a, i, b, j+1, c, k+1);
  }
}

vector<int> objedini(const vector<int>& a, const vector<int>& b) {
  vector<int> c(a.size() + b.size());
  objedini(a, 0, b, 0, c, 0);
  return c;
}


int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo prvi niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ucitavamo drugi niz
  int m;
  cin >> m;
  vector<int> b(m);
  for (int i = 0; i < m; i++)
    cin >> b[i];

  // objedinjavamo dva sortirana niza u treci
  vector<int> c = objedini(a, b);

  // ispisujemo rezultat
  for (int i = 0; i < m+n; i++)
    cout << c[i] << " ";
  cout << endl;

  return 0;
}
```

Задатак: Број парова датог збира

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Последњих k цифара степена

Написати програм којим се за дате природне бројеви a, n и k приказује последњих k цифара степена a^n.

Улаз: На стандардном улазу налазе се природни бројеви a ($a \leq 10^9$), n ($n \leq 10^9$) и k ($k \leq 9$), сваки у посебној линији.

Излаз: На стандарном излазу у једној линији приказати последњих (крајње десних) k цифара степена a^n.

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
2	192	10	000	3452	0576
13		5		20	
3		3		4	

Решење

Класичан начин да израчунамо степен a^n је применом итеративног алгоритма којим n пута вредност степена (иницијализовану на 1) множимо бројем a. Пошто нам је потребно само последњих k цифара броја аритметику вршимо по модулу 10^k. У задатку Операције по модулу видели смо да важи $(x \cdot y) \bmod m = (x \bmod m \cdot y \bmod m) \bmod m$. Зато на почетку број a можемо заменити његовим остатком при дељењу са 10^k. Ако након сваког множења текуће вредности степена са a извршимо и операцију израчунавања остатка при дељењу са 10^k, то не морамо радити пре наредног множења.

Нажалост, сложеност овог алгоритма је линеарна у односу на степен n, а пошто степен може бити прилично велики, овај алгоритам је неефикасан.

```
#include <iostream>

using namespace std;

long long stepen(long long a, long long n) {
  long long rez = 1;
  for (int i = 0; i < n; i++)
    rez *= a;
  return rez;
}

long long stepen(long long a, long long n, long long mod) {
  a = a % mod;
  long long rez = 1;
  for (int i = 0; i < n; i++)
    rez = (rez * a) % mod;
  return rez;
}

// ispisuje poslednjih k cifara broja x
void ispis(long long x, int k) {
    if (k == 0) return;
    ispis(x / 10, k - 1);
    cout << x % 10;
}

int main() {
  long long a, n, k;
  cin >> a >> n >> k;
  long long rez = stepen(a, n, stepen(10, k));
  ispis(rez, k);
  cout << endl;
  return 0;
}
```

Један корак ка ефикаснијој имплементацији може бити прелаз на рекурзивну имплементацију. Важи да је $a^0 = 1$, а да је $a^n = a \cdot a^{n-1}$. На основу овога се једноставно дефинише рекурзивна функција, при чему поново сву аритметику треба вршити по модулу. Алгоритам остаје неефикасан (сложености $O(n)$), а доводи и до додатних проблема у односу на итеративну имплементацију, јер је услед рекурзије меморијски захтевнији и за веће вредности n доводи до прекорачења системског стека.

```
#include <iostream>
```

```cpp
using namespace std;

// izracunava a ^ n
long long stepen(long long a, long long n) {
  if (n == 0)
    return 1;
  else
    return a * stepen(a, n-1);
}


// izracunava (a ^ n) % mod
long long stepen_(long long a, long long n, long long mod) {
  if (n == 0)
    return 1;
  else
    return (a * stepen_(a, n - 1, mod)) % mod;
}

long long stepen(long long a, long long n, long long mod) {
  return stepen_(a % mod, n, mod);
}


// ispisuje poslednjih k cifara broja x
void ispis(long long x, int k) {
  if (k > 0) {
    ispis(x / 10, k - 1);
    cout << x % 10;
  }
}

int main() {
  long long a, n;
  int k;
  cin >> a >> n >> k;
  ispis(stepen(a, n, stepen(10, k)), k);
  cout << endl;
  return 0;
}
```

Рекурзивну формулацију алгоритма је једноставно прилагодити тако да се дође до ефикасног алгоритма степеновања (тзв. алгоритма брзог степеновања). Основна идеја је да се за парне степене n примети да важи да је $a^n = (a^2)^{n/2}$. Излаз из рекурзије остаје $a^0 = 1$, а за непарне n се и даље примењује правило $a^n = a \cdot a^{n-1}$. Пошто се бар у сваком другом кораку вредност n смањи дупло, сложеност овог алгоритма је $O(\log n)$.

Размотримо како се израчунава 2^{10}.

$$2^{10} = 4^5 = 24 \cdot 4^4 = 4 \cdot 16^2 = 4 \cdot 256^1 = 4 \cdot 256 \cdot 256^0 = 4 \cdot 256 \cdot 1 = 1024$$

Наравно, овај алгоритам треба прилагодити тако да се после сваког множења израчунава вредност остатка при дељењу са 10^k.

```cpp
#include <iostream>

using namespace std;

// izracunava a ^ n
```

```
long long stepen(long long a, long long n) {
  if (n == 0)
    return 1;
  if (n % 2 == 0)
    return stepen(a * a, n / 2);
  else
    return a * stepen(a, n-1);
}

// izracunava (a ^ n) % mod
long long stepen_(long long a, long long n, long long mod) {
  if (n == 0)
    return 1;
  if (n % 2 == 0)
    return stepen_((a * a) % mod, n / 2, mod);
  else
    return (a * stepen_(a, n - 1, mod)) % mod;
}

long long stepen(long long a, long long n, long long mod) {
  return stepen_(a % mod, n, mod);
}

// ispisuje poslednjih k cifara broja x
void ispis(long long x, int k) {
  if (k > 0) {
    ispis(x / 10, k - 1);
    cout << x % 10;
  }
}

int main() {
  long long a, n;
  int k;
  cin >> a >> n >> k;
  ispis(stepen(a, n, stepen(10, k)), k);
  cout << endl;
  return 0;
}
```

У циљу превођења рекурзивне у итеративну имплементацију, функцију је могуће дефинисати и коришћењем репне рекурзије. Функција степеновања прима 3 параметра a, n и r, при чему у сваком рекурзивном позиву важи да је $a^n \cdot r$ има исту вредност. Ако у почетном рекурзивном позиву поставимо вредност $r = 1$, та иста вредност ће бити једнака траженој вредности $a_0^{n_0}$. Циљ нам је да у сваком кораку половимо n тј. да извршимо његово целобројно дељење са 2. Ако је n парно, тада је довољно квадрирати a, а r оставити непромењено. Ако је n непарно, тада је потребно додатно помножити r са a. Излаз из рекурзије представља случај $n = 0$ и тада функција може да врати r као резултат (пошто се вредност $a^n \cdot r$ не мења и једнака је почетној вредности $a_0^{n_0}$, када је $n = 0$, тада је $a^n \cdot r = r = a_0^{n_0}$.

$s(2, 10, 1) = s(4, 5, 1) = s(16, 2, 4) = s(256, 1, 4) = s(65536, 0, 1024) = 1024$

```
#include <iostream>

using namespace std;

// izracunava a ^ n
long long stepen_(long long a, long long n, long long rez) {
  if (n == 0)
    return rez;
  else
```

```
    return stepen_(a * a, n / 2, n % 2 == 0 ? rez : a * rez);
}

// izracunava a ^ n
long long stepen(long long a, long long n) {
  return stepen_(a, n, 1);
}

// izracunava (a ^ n) % mod
long long stepen_mod_(long long a, long long n, long long mod, long long rez) {
  if (n == 0)
    return rez;
  else
    return stepen_mod_((a * a) % mod, n / 2, mod,
                       n % 2 == 0 ? rez : (a * rez) % mod);
}

long long stepen_mod(long long a, long long n, long long mod) {
  return stepen_mod_(a % mod, n, mod, 1);
}

// ispisuje poslednjih k cifara broja x
void ispis(long long x, int k) {
  if (k > 0) {
    ispis(x / 10, k - 1);
    cout << x % 10;
  }
}

int main() {
  long long a, n;
  int k;
  cin >> a >> n >> k;
  ispis(stepen_mod(a, n, stepen(10, k)), k);
  cout << endl;
  return 0;
}
```

Брзо степеновање је могуће реализовати и итеративно. До итеративне имплементације се лако долази елиминисањем репне рекурзије. У сваком кораку n целобројно делимо са 2, а ако је n непарно и квадрирамо основу a, при чему ако је n непарно, тада резултат множимо са тренутном вредношћу основе. 2^10 се тада израчунава на следећи начин.

```
a      n      r
2     10      1
4      5      1
16     2      4
256    1      4
65536  0   1024
```

Инваријанта овог алгоритма је да се вредност $a^n \cdot r$ не мења током извршавања алгоритма. На почетку је $r = 1$, па је та вредност једнака траженој вредности a_0^n. На крају је $n = 0$, па је та вредност једнака $1 \cdot r$ тј. r. Дакле, вредност промењиве r на крају једнака је траженој.

```
#include <iostream>

using namespace std;

// izracunava a ^ n
long long stepen(long long a, long long n) {
```

```
  long long rez = 1;
  while (n > 0) {
    if (n % 2 == 1)
      rez *= a;
    n /= 2;
    a = a * a;
  }
  return rez;
}

// izracunava (a ^ n) % mod
long long stepen(long long a, long long n, long mod) {
  a = a % mod;
  long long rez = 1;
  while (n > 0) {
    if (n % 2 == 1)
      rez = (rez * a) % mod;
    n = n / 2;
    a = (a * a) % mod;
  }
  return rez;
}


// izracunava broj cifara broja n
int brojCifara(long long n) {
  int br = 0;
  do {
    n = n / 10;
    br++;
  }
  while (n > 0);
  return br;
}

// ispisuje poslednjih k cifara broja n (pod pretpostavkom da je on
// manji od 10^k)
void ispisi(long long n, int k) {
  int br = brojCifara(n);
  for (int i = 0; i < k - br; i++)
    cout << 0;
  cout << n << endl;
}


int main() {
  long long a, n;
  int k;
  cin >> a >> n >> k;
  ispisi(stepen(a, n, stepen(10, k)), k);
  return 0;
}
```

3.2 Репна рекурзија

Задатак: Збир n бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Задатак је могуће решити и помоћу репно-рекурзивне функције, која је алгоритамски веома слична итеративном алгоритму. Функција је репно-рекурзивна ако се након рекурзивног позива не врше даља израчунавања. Репна рекурзија је значајна због тога је многи компилатори програмских језика могу аутоматски претворити и једноставно превести у итерацију, чиме се добија и временски и меморијски ефикаснији алгоритам.

Претпоставимо да се учитавају редом бројеви 1, 2 и 3. Ако обележимо нашу рекурзивну функцију са z, вршиће се следећи рекурзивни позиви: $z(3) = z(3,0) = z(2,0+1) = z(2,1) = z(1,1+2) = z(1,3) = z(0,3+3) = z(0,6) = 6$.

```
#include <iostream>
using namespace std;

int ucitaj_i_saberi_n_brojeva(int n, int zbir) {
  if (n == 0)
    return zbir;
  int broj;
  cin >> broj;
  return ucitaj_i_saberi_n_brojeva(n-1, zbir + broj);
}

int ucitaj_i_saberi_n_brojeva(int n) {
  return ucitaj_i_saberi_n_brojeva(n, 0);
}

int main() {
  int n;
  cin >> n;
  cout << ucitaj_i_saberi_n_brojeva(n) << endl;
  return 0;
}
```

Задатак: Број и збир цифара броја

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Број и збир цифара могуће је одредити и репно-рекурзивним функцијама.

```
#include <iostream>
using namespace std;

void brojIZbirCifara_rek(int n, int& broj, int& zbir) {
  if (n > 0) {
    broj++;
    zbir += n % 10;
    brojIZbirCifara_rek(n / 10, broj, zbir);
  }
}

void brojIZbirCifara(int n, int& broj, int& zbir) {
  broj = 1; zbir = n % 10;
  brojIZbirCifara_rek(n / 10, broj, zbir);
}

int main() {
  int n;
```

```
  cin >> n;
  int broj, zbir;
  brojIZbirCifara(n, broj, zbir);
  cout << broj << " " << zbir << endl;
}
```

Задатак: Факторијел

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Задатак је могуће решити и помоћу репно-рекурзивне функције, која је алгоритамски веома слична итеративном алгоритму. Функција је репно-рекурзивна ако се након рекурзивног позива не врше даља израчунавања. Репна рекурзија је значајна због тога је многи компилатори програмских језика могу аутоматски претворити и једноставно превести у итерацију, чиме се добија и временски и меморијски ефикаснији алгоритам.

Дефинисаћемо рекурзивну функцију којој ћемо прослеђивати тренутну вредност броја i, број n чији се факторијел израчунава и тренутну вредност производа природних бројева мањих од i. У тренутку када се догоди да је $i > n$ тај производ је тражени факторијел. У супротном вршимо рекурзивни позив увећавајући бројач i за 1, и множећи производ бројем i.

На пример, ако је $n = 3$, вршиће се следећи низ израчунавања: $f(3) = f(1, 3, 1) = f(2, 3, 1) = f(3, 3, 2) = f(4, 3, 6) = 6$.

```
#include <iostream>

using namespace std;

int faktorijel(int i, int n, int f) {
  if (i > n)
    return f;
  return faktorijel(i+1, n, f*i);
}

int faktorijel(int n) {
  return faktorijel(1, n, 1);
}

int main() {
  int n;
  cin >> n;
  cout << faktorijel(n) << endl;
  return 0;
}
```

Задатак: Степен

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Задатак можемо решити и репно-рекурзивном функцијом, која одговара итеративном алгоритму. Функција прима број x, текући степен s и број n који одређује колико још пута број s треба помножити са x да би се израчунао тражени степен.

Ако функцију обележимо са st, приликом израчунавања вредности 2^3 вршиће се следећи рекурзивни позиви $st(2, 3) = st(2, 1, 3) = st(2, 2, 2) = (2, 4, 1) = st(2, 8, 0) = 8$.

```
#include <iostream>
#include <iomanip>

using namespace std;

double stepen(double x, double s, int n) {
  if (n == 0)
    return s;
  return stepen(x, x*s, n-1);
}

double stepen(double x, int n) {
  return stepen(x, 1.0, n);
}

int main() {
  double x;
  int n;
  cin >> x >> n;
  cout << fixed << showpoint << setprecision(5)
       << stepen(x, n) << endl;
  return 0;
}
```

Задатак: Једнакост растојања

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Тражена тачка је аритметичка средина унетих тачака.

Збир можемо израчунати и коришћењем репно-рекурзивне функције, која одговара итеративном алгоритму и коју ће компилатор вероватно оптимизовати тако што ће је аутоматски превести у итеративни извршиви програм. Функција прима збир обрађених бројева до тренутка њеног позива, као и број n преосталих бројева које треба учитати и сабрати. Када је $n = 0$, сви бројеви су сабрани и функција враћа тај акумулирани збир. У супротном се учитава нови број и функција се рекурзивно позива, умањујући број преосталих сабирака, а увећавајући збир за тај учитани број.

```
#include <iostream>
#include <iomanip>

using namespace std;

// ucitava n realnih brojeva i izracunava njihov zbir
double zbirUcitanihBrojeva(int n, double zbir) {
  if (n == 0)
    return zbir;
  double x;
  cin >> x;
  return zbirUcitanihBrojeva(n-1, zbir + x);
}

double zbirUcitanihBrojeva(int n) {
  return zbirUcitanihBrojeva(n, 0.0);
}


int main() {
```

```
  // broj tacaka
  int n;
  cin >> n;
  // zbir koordinata svih tacaka
  double s = zbirUcitanihBrojeva(n);

  // izracunavamo i ispisujemo prosek koordinata svih tacaka
  cout << fixed << setprecision(5) << showpoint << s / n << endl;
  return 0;
}
```

Задатак: Најнижа температура

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Задатак је могуће решити и помоћу репно-рекурзивне функције, која је алгоритамски веома слична итеративном алгоритму. Функција је репно-рекурзивна ако се након рекурзивног позива не врше даља израчунавања. Репна рекурзија је значајна због тога је многи компилатори програмских језика могу аутоматски претворити и једноставно превести у итерацију, чиме се добија и временски и меморијски ефикаснији алгоритам.

Дефинисаћемо функцију која одређује минимум између вредности свог првог прослеђеног параметра и вредности n бројева који се учитавају са стандардног улаза. Тај први параметар представља тренутну вредност минимума током итерације (и може се иницијализовати на вредност $+\infty$ тј. вредност највећег податка целобројног типа). Ако је $n = 0$, та тренутна вредност минимума је уједно и тражена вредност минимума свих бројева (до тада су сви они учитани и обрађени). У супротном се учитава наредни број, пореди се са вредношћу минимума и мањи од њих се прослеђује следећем рекурзивном позиву (као тренутна вредност минимума свих до тада учитаних бројева). Ако се редом учитавају бројеви $3, 8, 1, 2, 5$, извршиће се следеће израчунавање $Min(5) = Min(\infty, 5) = Min(\min(\infty, 3), 4) = Min(3, 4) = Min(\min(3, 8), 3) = Min(3, 3) = Min(\min(3, 1), 2) = Min(1, 2) = Min(\min(1, 2), 1) = Min(1, 1) = Min(\min(1, 5), 0) = Min(1, 0) = 1$.

```
#include <iostream>
#include <algorithm>
#include <limits>

using namespace std;

int Min(int m, int n) {
  if (n == 0)
    return m;
  int t;
  cin >> t;
  return Min(min(m, t), n-1);
}

int Min(int n) {
  return Min(numeric_limits<int>::max(), n);
}

int main() {
  int n;
  cin >> n;
  cout << Min(n) << endl;
  return 0;
}
```

Задатак: Други на ранг листи

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Задатак можемо решити и репном рекурзијом (коју неки компилатори аутоматски оптимизују и претварају у итерацију). Дефинисаћемо рекурзивну функцију која прима два највећа елемента у делу низа који је обрађен пре позива те функције и број n преосталих елемената које треба обрадити. Ако је $n = 0$ цео низ је обрађен и функција може да врати дотадашњих други елемент по величини. У супротном, учитава се наредни елемент x и врши се рекурзивни позив за преосталих $n - 1$ елемената низа, при чему се по потреби ажурирају два до тада највећа елемента. Ако је учитани елемент x већи од дотадашњег првог максимума, онда је он нови први максимум, а дотадашњи први максимум је нови други максимум. У супротном, ако је учитани елемент x већи од дотадашњег другог максимума, тада се први максимум не мења, а елемент x постаје нови други максимум. У супротном се не мењају ни први ни други максимум.

```
#include <iostream>

using namespace std;

int drugi(int n, int prviMaks, int drugiMaks) {
  if (n == 0)
    return drugiMaks;
  int x;
  cin >> x;
  if (x > prviMaks)
    return drugi(n-1, x, prviMaks);
  else if (x > drugiMaks)
    return drugi(n-1, prviMaks, x);
  else
    return drugi(n-1, prviMaks, drugiMaks);
}

int drugi(int n) {
  return drugi(n, -1, -1);
}


int main() {
  ios_base::sync_with_stdio(false);

  // broj elemenata serije
  int n;
  cin >> n;

  // ispisujemo vrednost drugog maksimuma
  cout << drugi(n) << endl;

  return 0;
}
```

Задатак: Прва негативна температура

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

У задатку Негативан број питали смо се да ли међу бројевима постоји број који је негативан, а у овом задатку тражимо мало више од тога - тражимо да одредимо прву позицију на којој се јавља неки негативан број. Решење опет можемо решити алгоритмом линеарне претраге, који можемо имплементирати на било који од начина приказаних у задатку Негативан број.

Задатак се може решити дефинисањем репно-рекурзивне функције, која је веома блиска итеративном алгоритму. Функција прима текућу позицију i и укупан број елемената n и треба да учита $n - i + 1$ елемената (у интервалу позиција $[i, n]$) и одреди позицију првог негативног међу њима (или да врати -1 ако међу њима нема негативних). У иницијалном позиву ћемо поставити вредност $i = 1$, да би функција одредила позицију првог негативног елемента у интервалу позиција $[1, n]$. Ако је $i > n$, тада нема елемената, па функција може да врати -1. У супротном се учитава први елемент (он је на позицији i). Ако је он негативан, то је уједно и први негативан и функција враћа позицију i. У супротном се врши рекурзивни позив да би се одредила позиција првог негативног у делу низа између позиција $i + 1$ и n.

```
#include <iostream>
using namespace std;

int prviNegativan(int i, int n) {
  if (i > n)
    return -1;
  int x; cin >> x;
  return x < 0 ? i : prviNegativan(i+1, n);
}

int prviNegativan(int n) {
  return prviNegativan(1, n);
}

int main() {
  int n;
  cin >> n;
  cout << prviNegativan(n) << endl;
  return 0;
}
```

Задатак: Последња негативна температура

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Задатак можемо решити и репно-рекурзивном функцијом. Функција прима позицију p последњег негативног елемента међу елементима учитаним пре него што је је функција позвана, позицију i тренутног елемента који ће евентуално бити учитан током позива функције и укупан број елемената n. Ако је $i > n$ (пошто се позиције броје од 1), тада су већ обрађени сви елементи и резултат је позиција p. У супротном се учитава тренутни елемент x и рекурзивно се позива функција увећавајући бројач i и ажурирајући позицију p, ако је x негативан (тада се уместо p у рекурзивни позив шаље вредност i).

У првом позиву прослеђујемо вредност $p = -1$ (јер док није учитан ниједан елемент, међу њима нема негативних), $i = 1$ (јер бројање позиција почиње од 1) и n.

```
#include <iostream>

using namespace std;

int poslednjiNegativan(int p, int i, int n) {
  if (i > n)
    return p;
  int x; cin >> x;
  return poslednjiNegativan(x < 0 ? i : p, i+1, n);
}

int poslednjiNegativan(int n) {
  return poslednjiNegativan(-1, 1, n);
}
```

```
int main() {
  int n;         //  broj dana koji se analiziraju
  cin >> n;
  cout << poslednjiNegativan(n) << endl;
  return 0;
}
```

Задатак: Просек одличних

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

double prosekOdlicnih(int n, double zbirProsekaOdlicnih, int brojOdlicnih) {
  if (n == 0)
    return zbirProsekaOdlicnih / brojOdlicnih;
  else {
    double tekuciProsek;
    cin >> tekuciProsek;
    if (tekuciProsek >= 4.5)
      return prosekOdlicnih(n-1, zbirProsekaOdlicnih + tekuciProsek,
                                 brojOdlicnih + 1);
    else
      return prosekOdlicnih(n-1, zbirProsekaOdlicnih, brojOdlicnih);
  }
}

double prosekOdlicnih(int n) {
  return prosekOdlicnih(n, 0.0, 0);
}


int main() {
  int n;     // ukupan broj ucenika
  cin >> n;
  double prosek = prosekOdlicnih(n);
  if (isfinite(prosek))
    cout << fixed << setprecision(2) << showpoint
         << prosek << endl;
  else
    cout << "-" << endl;

  return 0;
}
```

3.3 Извођење итеративних алгоритама из рекурзивних

Задатак: Грејов код

Грејов кôд реда n подразумева ређање свих n-тоцифрених бинарних записа тако да се свака два суседна записа разликују тачно у једном биту (при чему ово важи и за први и последњи запис, тако да се може сматрати да

су сви записи поређани у круг).

Грејов кôд дужине 0 садржи само један елемент и то празну ниску. Грејов кôд дужине $n+1$ се може добити од кода дужине n тако што се испред сваког броја у коду дужине n допише цифра 0, затим се редослед елемената у коду дужине n обрне и на сваком броју се на почетак допише цифра 1 и два тако добијена низа бројева се споје. Нпр. Грејов кôд реда 2 је

```
00
01
11
10
```

На основу претходног поступка добијамо Грејов кôд реда 3.

```
0 00  |                k
0 01  |                0:  000
0 11  |                1:  001
0 10  v                2:  011
             tj.       3:  010
1 10  ^                4:  110
1 11  |                5:  111
1 01  |                6:  101
1 00  |                7:  100
```

Напиши програм који за дату дужину кода n и дату позицију k ($0 \leq k < 2^n$) одређује бинарни број који се налази на позицији k у коду дужине n.

Улаз: Са стандардног улаза се учитава дужина кода n ($1 \leq n \leq 32$) и позиција k ($0 \leq k < 2^n$).

Излаз: На стандардни излаз исписати тражени бинарни број.

Пример 1

Улаз	*Излаз*
3	011
2	

Пример 2

Улаз	*Излаз*
30	10011001010111101011010000000
999999999	

Решење

Дефинишимо функцију која одређује k-ти по реду запис Грејовог кода дужине 2^n (подразумеваћемо да је $0 \leq k < 2^n$). Њу је једноставно дефинисати рекурзивно.

Ако је n нула, резултат је празна ниска. У супротном треба израчунати неки елемент Грејовог кода дужине $n-1$ и затим га допунити слева нулом или јединицом. Треба разликовати случај елемената у првој и у другој половини листе кодова. Пошто укупно има 2^n кодова, елементи у првој половини су на позицијама $0 \leq k < 2^{n-1}$, док су елементи у другој половини на позицијама $2^{n-1} \leq k < 2^n$.

- Када је $0 \leq k < 2^{n-1}$, тада се враћа k-ти елемент Грејовог кода дужине 2^{n-1} допуњен почетном нулом.
- Када је $2^{n-1} \leq k < 2^n$ тада се враћа $2^n - 1 - k$-ти елемент Грејовог кода дужине 2^{n-1} допуњен почетном јединицом. Изразом $2^n - 1 - k$ се позиција k своди у распон $[0, 2^{n-1})$ и уједно се обрће редослед бројева. Наиме, операцијом $k - 2^{n-1}$ вршимо редукцију интервала $[2^{n-1}, 2^n)$ на интервал $[0, 2^{n-1})$. Генерално, приликом обртања редоследа елемената, свака позиција p у интервалу $[0, m)$ се пресликава у позицију $m-p-1$ (позиција 0 се слика у $m-1$, док се $m-1$ слика у 0). Стога се приликом обртања интервала $[0, 2^{n-1})$ позиција $k-2^{n-1}$ слика у $2^{n-1}-(k-2^{n-1})-1$, но то је једнако 2^n-1-k.

Израчунавање степена двојке најједноставније се врши битовским операцијама (при чему треба обратити пажњу на потенцијално прекорачење).

Резултат можемо представити у облику ниске карактера. Иако дописивање карактера на почетак ниске може бити неефикасна операција, с обзиром на то да су ниске са којима радимо прилично кратке (најдужа има 32 карактера), о том не морамо да бринемо.

```
string grej(unsigned n, unsigned k) {
   if (n == 0)
      return "";
   if (k < (1u << (n - 1)))
      return "0" + grej(n - 1, k);
```

```
   else
      return "1" + grej(n - 1, (1ul << n) - 1 - k);
}
```

Функцију можемо реализовати и итеративно. Током итерације у променљивој `rez` налазиће се префикс траженог броја дужине $n_0 - n$, где је n_0 почетна, а n тренутна дужина кода. У зависности од тога да ли је текућа позиција k испод или изнад средине, префикс ћемо проширивати (здесна) нулом или јединицом, и уместо рекурзивног позива вредности k и n ћемо ажурирати вредностима које би се наводиле у рекурзивном позиву.

```
string grej(unsigned n, unsigned k) {
   string rez = "";
   while (n > 0) {
     if (k < 1u << (n-1))
        rez = rez + "0";
     else {
        rez = rez + "1";
        k = (1ul << n) - 1 - k;
     }
     n--;
   }
   return rez;
}
```

Напоменимо и да се тражени број у Грејовом коду може израчунати и директно, ако се представи у облику неозначеног броја изразом `k ^ (k >> 1)` (због водећих нула дужина n тада није битна). Овим се врши ексклузивна дисјункција позиције k и броја добијеног шифтовањем битова те позиције удесно. Неозначени број можемо затим претворити у ниску карактера и издвојити њен суфикс дужине n.

```
string grej(unsigned n, unsigned k) {
  return bitset<32>(k ^ (k >> 1)).to_string().substr(32 - n, n);
}
```

Задатак: Абацаба

Низ слова $ABACABADABACABAEABACABADABACABAFABACA$... се може формирати на следећи начин:

1. Низ је на почетку празан.
2. На низ се допише прво велико слово енглеског алфабета које се не појављује у формираном делу низа, а иза тог слова се понове сва слова која су се појавила пре њега.
3. Корак 2 се понови потребан број пута

Тако после прве примене корака 2 добијамо низ A, после друге низ ABA, после треће низ $ABACABA$ итд.

Одредити слово које се појављује на n-том месту у низу, бројећи места од 1. Редослед слова у енглеском алфабету је ABCDEFGHIJKLMNOPQRSTUVWXYZ.

Улаз: Један природан број мањи од 67 108 864.

Излаз: Једно велико слово енглеског алфабета.

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
8	D	65	A	100	C

Решење

Решење грубом силом је да се у меморији креира цео низ карактера и да се затим прочита карактер са одговарајуће позиције. Ово решење троши превише меморије, а и времена док се дугачак низ карактера не изгради.

```
string s = "";
char slovo = 'A';
while (s.size() <= n - 1) {
  s = s + slovo + s;
```

```
  slovo++;
}
cout << s[n-1] << endl;
```

Задатак се може решити без креирања дугачке ниске карактера, ако пажљиво проучимо правилност по којој се слова појављују. Прво слово A се налази на месту 1, прво слово B на месту 2, прво слово C на месту 4, прво слово D на месту 8 итд., па се може наслутити да се прва појављивања слова налазе на местима која су степени броја 2.

Заиста, ако дужину низа карактера пре уметања новог слова абецеде обележимо са d_k, важи да је $d_0 = 0$ (пре уметања слова A не налази се ниједан карактер) и да је $d_{k+1} = 2d_k + 1$ (пре уметања новог слова налази се ниска која је добијена тако што је претходно слово спојено са два појављивања ниске која се појављивала пре тог претходног слова). Зато је $d_k = 2^k - 1$ (важи да је $2^0 - 1 = 0$ и да је $2^{k+1} - 1 = 2 \cdot (2^k - 1) + 1$), док је прва позиција слова k (ако се слова броје од један) једнака 2^k.

Дакле, ако је унети број n неки степен двојке $n = 2^k$, онда је у питању место на ком се слово први пут појављује и важи да је $k = \log_2 n$ ($\log_2 n$ означава број k такав да је $2^k = n$, нпр. $\log_2 32 = 5$, јер је $2^5 = 32$). Знајући k слово лако можемо одредити сабирајући k са ASCII/UNICODE кодом слова A.

У супротном проблем можемо свести на проблем мање димензије. Нека је $2^k < n < 2^{k+1}$, тј. нека је k највећи степен двојке мањи од n. Карактер који тражимо налази се, дакле, иза позиције 2^k у делу низа који је добијен копирањем дела низа испред позиције 2^k. Зато је n-ти карактер у целом низу једнак $(n - 2^k)$-том карактеру у копираном делу, а пошто део који се копира почиње на почетку низа, једнак је $(n - 2^k)$-том карактеру у целом низу.

Овим смо описали рекурзивни поступак којим ефикасно можемо доћи до решења.

$$f(n) = \begin{cases} \log_2 n, \text{за } n = 2^k \\ f(n - 2^k), \text{за } 2^k < n < 2^{k+1} \end{cases} .$$

На пример, $f(23) = f(23 - 16) = f(7) = f(7 - 4) = f(3) = f(3 - 2) = f(1) = 0$, па се на месту 23 налази слово A. Слично, на пример, важи да је $f(40) = f(40 - 32) = f(8) = 3$, па се на месту 40 налази слово D.

На основу претходне дефиниције би се могла имплементирати рекурзивна функција (за то би било потребно испитати да ли је дати број степен броја 2, наћи логаритам таквог броја, и наћи највећи степен броја 2 од ког је дати број већи или једнак). Ипак, за тим нема потребе. Анализирајући рад такве рекурзивне функције можемо унапред закључити шта ће њен резултат бити, без потребе за њеним извршавањем. Претпоставимо да знамо бинарни запис броја n тј. да знамо да се број n представља као збир неких степенова двојке $2^{s_m} + 2^{s_{m-1}} + \ldots + 2^{s_0}$. Током рекурзије од броја ће се одузимати један по један степен двојке, све док не остане само 2^{s_0} и тада ћемо знати да је $k = s_0$. Дакле важи да је резултат једнак најмањем степену двојке који учествује разлагању броја на збир степенова двојке тј. да је тражени број k једнак позицији најдешње јединице у бинарном запису броја (претпостављамо да се позиције броје од 0, здесна). До траженог резултата је могуће доћи дељењем броја са 2_0^s, тј. узастопним дељењем броја са 2 све док последњи сабирак не постане 1, тј. све док је број паран (то ће се догодити тачно s_0 пута).

```
int k = 0;
while (n % 2 == 0) {
  n /= 2;
  k++;
}
cout << (char)('A' + k) << endl;
```

Сличан резултат можемо добити и баратајући директно са интерним записом броја у рачунару, коришћењем битовских оператора. Позицију крајње десне јединице једноставно можемо израчунати шифтовањем (померањем) бинарног записа броја удесно за једно место све док последња цифра не постане једнака 1 (последњу цифру можемо испитати битовском конјункцијом са 1).

```
int k = 0;
while ((n & 1) == 0) {
  n >>= 1;
  k++;
```

```
}
cout << (char)('A' + k) << endl;
```

Савремени хардвер често поседује и инструкцију *count trailing zeroes* којом се одређује број нула иза последње јединице у бинарном запису (што је баш позиција последње јединице). Иако програмски језици обично не стандардизују приступ овој операцији, неки компилатори нуде подршку за то (на пример, ако се користи GCC, може се употребити функција `__builtin_ctz`, а ако се користи Microsoft Visual C++, може се употребити функција `_BitScanReverse`).

```
cout << (char)('A' + __builtin_ctz(n)) << endl;
```

Задатак: Морзеов низ

Низ $1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, \ldots$, који се састоји од нула и јединица, гради се на следећи начин: први елемент је 1; други се добија логичком негацијом првог $NOT(1) = 0$, трећи и четврти логичком негацијом претходна два $NOT(1) = 0$, $NOT(0) = 1$, пети, шести, седми и осми логичком негацијом прва четири – добија се $0, 1, 1, 0$ итд. Дакле, кренувши од једночланог сегмента 1, сваком почетном сегменту који је дужине 2^k (k узима вредности $0, 1, 2, \ldots$) дописује се сегмент исте дужине добијен логичком негацијом свих елемената почетног сегмента. За задато n одредити n-ти члан низа (бројање креће од 1).

Улаз: У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 10^9$).

Излаз: На стандарном излазу приказати цифру (0 или 1) на позицији n

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
15	0	1234	0	12345678	1

Решење

Формирање целог низа

Формулација задатка сугерише решење у коме би се почев од задатог првог елемента $x_1 = 1$ редом формирали сегменти по задатим правилима (x_2), (x_3, x_4), $(x_5 \ldots x_8)$, $(x_9 \ldots x_{16})$, $(x_{17} \ldots x_{32})$, $\ldots$. Дакле, елементи текућег сегмента се формирају негацијом претходно формираних елемената $x_1 \ldots x_k$, који су на растојању k, где је k степен броја 2 (увећава се дупло након сваког преписаног сегмента).

Сложеност овог приступа је $O(n)$.

Ово уписивање се може реализовати помоћу угнежђених петљи где унутрашња петља дуплира садржај низа, а спољашња контролише колико пута садржај треба дуплирати (додатно осигуравајући да се уписивање прекине када се попуни потребних n елемената низа).

```
// Morzeov niz
vector<bool> a(n+1);
a[1] = true;
int upisano = 1; // broj upisanih elemenata
int duzina = 1; // duzina segmenta koji se trenutno negira
while (upisano < n) {
  // negiramo segment trenutne duzine
  // prekidamo petlju ranije ako tokom toga dostignemo n upisanih elemenata
  for (int i = 1; i <= duzina && upisano < n; i++) {
    a[duzina + i] = !a[i];
    upisano++;
  }
  // naredni segment koji treba negirati je duplo duzi
  duzina *= 2;
}

// upisano je n elemenata niza, pa ocitavamo rezultat
cout << (a[n] ? "1" : "0") << endl;
```

Низ се може попунити и помоћу само једне петље у којој се врши преписивање елемената док се не упише њих n тако што се на место i преписује елемент са позицијје $i - k$, увећавајући степен двојке k када се цео

претходни подсегмент препише.

```
// Morzeov niz
vector<bool> a(n+1);
a[1] = true;
// rastojanje elemenata koji se negiraju
int k = 1;
// popunjavamo niz zakljucno sa pozicijom n
for (int i = 2; i <= n; i++) {
  // negiramo odgovarajuci element
  a[i] = !a[i - k];
  // negirali smo ceo segment, pa prelazimo na sledeci
  if (i == 2 * k)
    k *= 2;
}
// upisano je n elemenata niza, pa ocitavamo rezultat
cout << (a[n] ? "1" : "0") << endl;
```

Веза између одговарајућих чланова

Директна решења задатка подразумевају формирање свих елемената низа који претходе n-том члану. Да би се израчунао 2001. члан мора се израчунати свих 2000 претходних чланова.

Уочимо следеће, како бисмо проблем решили без коришћења низа и без израчунавања свих чланова низа који претходе n-том: формирање елемената сегмента на основу претходног поступка, значи да се сваки пут дужина низа удваја, тј. представља степен броја 2.

- Приликом негирања почетног сегмента добијамо да је $x_2 = NOT(x_1)$. У овом случају важи да је $x_n = NOT(x_{n-1})$.
- Негирањем наредног сегмента добијамо да је $x_3 = NOT(x_1)$ и $x_4 = NOT(x_2)$. У овом случају важи да је $x_n = NOT(x_{n-2})$.
- Након тога добијамо да је $x_5 = NOT(x_1)$, $x_6 = NOT(x_2)$, $x_7 = NOT(x_3)$ и $x_8 = NOT(x_4)$. У овом случају важи да је $x_n = NOT(x_{n-4})$.

Дакле, за $n > 1$ важи рекурентна формула $x_n = NOT(x_{n-m})$, где је m - максимални степен броја 2, који је строго мањи од n. Ова рекурентна формула омогућава веома ефикасно израчунавање траженог члана низа. На пример,

$x_{15} = NOT(x_{15-8}) = NOT(x_7) = NOT(NOT(x_{7-4})) = x_{7-4} = x_3 = NOT(x_{3-2}) = NOT(x_1) = NOT(1) = 0$.

Тражени број се добија у неколико корака и за веће вредности броја n. На пример, за $n = 2001$:

$x_{2001} = NOT(x_{2001-1024}) = NOT(x_{977}) = x_{977-512} = x_{465} = NOT(x_{465-256}) = NOT(x_{209}) = x_{209-128} = x_{81} = NOT(x_{81-64}) = NOT(x_{17}) = x_{17-16} = x_1 = 1$.

Рекурентна формула нам указује да решење можемо веома једноставно реализовати уз помоћу рекурзивне функције. У имплементацији се користи помоћна функција за одређивање траженог степена двојке (ову функцију је могуће имплементирати и ефикасније, коришћењем операција над битовима, међутим и ова једноставна имплементација је сасвим довољна).

Пошто се сваки елемент низа израчунава на основу неког из претходне половине низа, у сваком кораку се n бар полови, тако да је сложеност овог приступа $O(\log n)$.

```
// vraca najveci stepen od 2, koji je manji od n
int MaxStepen2(int n) {
  int max = 1;
  while (max * 2 < n)
    max *= 2;
  return max;
}

// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
```

```
int Morzeov(int n) {
  if (n == 1)
    return 1;
  return !Morzeov(n - MaxStepen2(n));
}
```

Рекурзију је могуће једноставно елиминисати и до решења можемо доћи и итеративно тако што ћемо полазећи од $x = 1$, при сваком преласку на индекс настао умањењем за највећи степен броја негирати текућу вредност x.

```
// vraca najveci stepen od 2, koji je strogo manji od n
int MaxStep2(int n) {
  int max = 1;
  while (max * 2 < n)
    max *= 2;
  return max;
}

// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
int Morzeov(int n) {
  int x = 1; // prvi clan niza
  while (n > 1) {
    n -= MaxStep2(n);
    x = !x;
  }
  return x;
}
```

Приметимо да се током претходно описаног поступка уклања један по један бит броја, све док не остане само један бит (тј. док број не постане степен двојке). Након тога се тај бит помера за једно место надесно, све док број не постане 1.

На пример, низ

$$x_{44} = NOT(x_{12}) = x_4 = NOT(x_2) = x_1 = 1$$

се бинарно може представити помоћу

$$x_{101100} = NOT(x_{001100}) = x_{000100} = NOT(x_{000010}) = x_{000001} = 1.$$

Ова друга фаза би се могла избећи, а имплементација поједноставити ако би бројање позиција било од 0, а не од 1 (што лако можемо постићи ако одмах након учитавања умањимо вредност n за 1). Тада би важило да је $x_1 = NOT(x_0)$, затим $x_2 = NOT(x_0)$, $x_3 = NOT(x_1)$, затим $x_4 = NOT(x_0)$, $x_5 = NOT(x_1)$, $x_6 = NOT(x_2)$ и $x_7 = NOT(x_3)$. Разлика је, дакле, у томе што се од сваког индекса одузима највећи степен двојке који је већи или једнак од датог броја (разлика је значајна баш када је индекс који тренутно разматрамо степен двојке). У тој варијанти бисмо уместо x_{44} рачунали x_{43} и важило би

$$x_{43} = NOT(x_{11}) = x_3 = NOT(x_1) = x_0 = 1$$

тј. бинарно

$$x_{101011} = NOT(x_{001011}) = x_{000011} = NOT(x_{000001}) = x_{000000} = 1.$$

Дакле, на овај начин се у сваком кораку уклања један бит броја, све док број не постане 0, негирајући сваки пут резултујући бит. Увид који нас доводи до лепше имплементације је то да ће се исти резултат добити и када се битови уклањају један по један, кренувши од битова најмање, уместо од битова највеће тежине. Уклањање последњег бита 1 у бинарном запису броја `n` се може урадити веома ефикасно коришћењем израза `n & (n - 1)`.

Можемо приметити да се у овом решењу потврђује да је бројање од 0 уместо од 1 природније и да има лепша математичка својства.

```
// prilagodjavamo brojanje tako da krene od 0 umesto od 1
n--;

int x = 1;
while (n) {
  // uklanjamo poslednji bit iz binarnog zapisa broja
  n = n & (n-1);
  x = !x;
}

cout << x << endl;
```

На крају, можемо увидети да је битан само број јединица у бинарном запису броја $n - 1$. Ако је тај број паран, резултат је 1, а ако је непаран, резултат је 0. Постоје ефикасни алгоритми да се тај број израчуна, а неки рачунари имају и уграђену хардверску инструкцију за то. Иако програмски језик C++ не стандардизује приступ тој инструкцији, ако се користи компилатор GCC, могуће је употребити функцију `__builtin_popcount`, а ако се користи Microsoft Visual C++, могуће је употребити функцију `__popcnt`.

```
// prilagodjavamo brojanje tako da krene od 0 umesto od 1
n--;
// ispitujemo parnost ukupnog broja bitova jednakih 1 u binarnom
// zapisu broja n
cout << !(__builtin_popcount(n) & 1) << endl;
```

Задатак: Избацивање цифара на све начине

Напиши програм који за дати природан број n одређује збир свих бројева који се могу добити избацивањем неких цифара броја n.

Улаз: Са стандардног улаза се учитава број који може да има највише 1000 цифара.

Излаз: На стандардни излаз исписати тражени збир.

Пример

Улаз	*Излаз*
123	177

Објашњење

$123 + 12 + 13 + 23 + 1 + 2 + 3 + 0 = 177$.

Решење

До решења можемо доћи индуктивно-рекурзивном конструкцијом. Ако је број једнак нули, тада је тражени збир једнак нули. У супротном се тај број може разложити на своју последњу цифру и цифре које јој претходе (нпр. број 1234 се може разложити на број 123 и цифру 4). Претпоставимо да умемо да одредимо тражени збир за број коме је уклоњена последња цифра и размотримо како се комбинацијом тог броја и последње цифре може добити тражени збир. У текућем примеру, претпостављамо, дакле, да умемо да одредимо тражени збир за број 123. На сваки од сабирака који учествује у том збиру можемо додати четворку здесна. Сваки од тих сабирака се добија тако што се оригинални број помножи са 10 и дода се 4. Њихов се збир зато може добити тако што се полазни збир (177) помножи са 10 и затим увећа за онолико четворки колико има сабирака (у текућем примеру их има 8). Пошто су овим покривене све могућности, укупан збир се добија сабирањем полазног и овако трансформисаног збира.

```
177     177*10 + 8*4 = 1802          177 + 1802 = 1979

123     1234
12      124
13      134
23      234
1       14
```

```
2       24
3       34
0       4
```

У општем случају, дакле, добијени збир префикса множимо са 10 и увећавамо производом последње цифре и броја бројева који се добијају прецртавањем цифара тог префикса. Тај број није тешко израчунати јер је прилично очигледно да k-тоцифрен број може да генерише 2^k бројева (свака од k цифара може бити или прецртана или непрецртана). Зато можемо ојачати индуктивну хипотезу и направити функцију која уз тражени збир враћа још додатно и број цифара броја (или још боље, одговарајући степен двојке).

Дакле, проблем једноставно можемо решити тако што дефинишемо рекурзивну функцију која прима број n а враћа тражени збир за тај број n и одговарајући степен двојке. Више вредности функција може да врати помоћу својих излазних параметара.

Додатни проблем представља величина бројева који су допуштени на улазу, тако да је јасно да имплементација алгоритма са библиотечким типовима података није исправна. Стога имплементација која користи само основне бројевне типове података не ради коректно за све могуће улазе (то се може решити коришћењем великих бројева).

```
void Zbir(int n, int& zbir, int& b) {
  if (n == 0) {
    zbir = 0;
    b = 1;
  } else {
    int zbirR, brojR;
    Zbir(n / 10, zbirR, brojR);
    zbir = zbirR +  zbirR*10 + brojR * (n % 10);
    b = brojR * 2;
  }
}
```

Овакве рекурзије је веома једноставно ослободити се и алгоритам се лако може имплементирати и итеративно. Размотримо како се извршава рекурзивни позив за аргумент $n = 123$.

```
Zbir(123)             177, 8
    Zbir(12)          15, 4
       Zbir(1)        1, 2
          Zbir(0)     0, 1
```

Примећујемо, дакле, да се израчунавања врше у повратку кроз рекурзију и то тако што се вредност претходног збира и претходног степена двојке ажурирају на основу описаног поступка. Исто израчунавање се може описати и итеративним поступком у ком се промењиве иницијализују на 0 и 1, а затим током итерације ажурирају коришћењем једне по једне цифре полазног броја, с лева надесно. Пролазак кроз цифре броја у том редоследу једноставнији је ако се број представи као ниска карактера.

```
int zbir = 0, b = 1;
for (char c : broj) {
  zbir += 10*zbir + b * (c - '0');
  b *= 2;
}
cout << zbir << endl;
```

Да бисмо добили потпуно исправно решење морамо користити рад са великим бројевима.

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

vector<int> saberi(const vector<int>& a, const vector<int>& b) {
  vector<int> rez;
  int p = 0;
```

```
  for (size_t i = 0; i < a.size() || i < b.size(); i++) {
    int c = (i < a.size() ? a[i] : 0) +
            (i < b.size() ? b[i] : 0) + p;
    rez.push_back(c % 10);
    p = c / 10;
  }
  if (p > 0)
    rez.push_back(p);
  return rez;
}

vector<int> mnoziBrojem(const vector<int>& a, int b) {
  vector<int> rez;
  int p = 0;
  for (size_t i = 0; i < a.size(); i++) {
    int c = a[i] * b + p;
    rez.push_back(c % 10);
    p = c / 10;
  }
  if (p > 0)
    rez.push_back(p);
  return rez;
}

vector<int> ucitajBroj(const string& broj) {
  vector<int> rez(broj.size());
  for (int i = broj.size() - 1; i >= 0; i--)
    rez.push_back(broj[i] - '0');
  return rez;
}

string ispisiBroj(const vector<int>& broj) {
  string s;
  for (int i = broj.size() - 1; i >= 0; i--)
    s.push_back('0' + broj[i]);
  return s;
}


int main() {
  string broj; cin >> broj;
  vector<int> zbir; zbir.push_back(0);
  vector<int> b; b.push_back(1);
  for (char c : broj) {
    zbir = saberi(zbir, saberi(mnoziBrojem(zbir, 10),
                               mnoziBrojem(b, c - '0')));
    b = mnoziBrojem(b, 2);
  }
  cout << ispisiBroj(zbir) << endl;
  return 0;
}
```

Задатак се много лакше може решити ако се користи библиотечки тип података за рад са великим бројевима. На жалост, та функционалност не постоји у стандардној библиотеци језика C++, али су доступне квалитетне библиотеке отвореног кода које се могу искористити (нпр. GNU Multiprecision Library - GMP).

Задатак: Не садрже цифру 3

Напиши програм који одређује колико природних бројева из интервала $[0, n]$ не садрже цифру 3 у свом декадном запису.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 2 \cdot 10^9$).

Излаз: У првој линији стандардног излаза приказати тражени резултат.

Пример 1

Улаз	*Излаз*
15	14

Објашњење

У интервалу $[0, 15]$ постоји 16 бројева, а бројеви 3 и 13 једини садрже цифру 3.

Пример 2

Улаз

```
999
```

Излаз

```
729
```

Пример 3

Улаз

```
12345
```

Излаз

```
8262
```

Решење

Бројање бројева који не садрже цифру 3

Задатак можемо решити тако што за сваки број од 0 до n проверимо да ли садржи цифру 3, и ако не садржи увећамо бројач бројева који не садрже цифру 3 (тај бројач у почетку иницијализујемо на нулу). У том решењу примењује се алгоритам одређивања броја елемената серије који задовољавају дати услов, тј. алгоритам бројања филтриране серије. Тај алгоритам је описан, на пример, у задатку Просек одличних. Проверу да ли број садржи цифру 3 можемо реализовати у посебној функцији.

```
bool sadrziCifru3(int n) {
  do {
    if (n % 10 == 3)
      return true;
    n /= 10;
  } while (n > 0);
  return false;
}

int main() {
  int n;
  cin >> n;
  int br = 0;
  for (int i = 0; i <= n; i++)
    if (!sadrziCifru3(i))
      br++;
  cout << br << endl;
  return 0;
}
```

Ефикасније израчунавање броја бројева

Задатак можемо решити и на много ефикаснији начин (али је решење у том случају доста комплексније). Нека $f(a, b)$ означава број бројева из интервала $[a, b]$ који у свом декадном запису не садрже цифру 3, а $f_0(n)$ број таквих бројева из интервала $[0, n]$. Размотримо пример у коме желимо да израчунамо вредност $f_0(4251)$. Све бројеве из интервала $[0, 4251]$ можемо поделити у неколико група тј. подинтервала. Важи да је

$$[0, 4251] = [0, 999] \cup [1000, 1999] \cup [2000, 2999] \cup [3000, 3999] \cup [4000, 4251].$$

Зато је

$$f_0(4251) = f(0, 999) + f(1000, 1999) + f(2000, 2999) + f(3000, 3999) + f(4000, 4251).$$

Важи да је $f(0, 999) = f_0(999)$. Такође, важи и да је $f(1000, 1999)$ једнак броју $f(0, 999)$ тј. $f_0(999)$. Заиста, између интервала $[0, 999]$ и $[1000, 1999]$ може се успоставити бијекција таква да слика у свом декадном запису садржи цифру 3 ако и само ако њен оригинал у свом декадном запису садржи цифру 3. Слично, важи и да је $f(2000, 2999) = f_0(999)$, док је $f(3000, 3999) = 0$ јер сви бројеви у интервалу $[3000, 3999]$ садрже цифру 3. На крају, важи и да је $f(4000, 4251)$ једнако $f(0, 251)$ тј. $f_0(251)$. Зато је

$$f_0(4251) = 3 \cdot f_0(999) + f_0(251).$$

Дакле, ако знамо бројеве $f_0(999)$ и $f_0(251)$ тада можемо израчунати и број $f_0(4251)$.

Иста техника се може применити и на израчунавање бројева $f_0(999)$ и $f_0(251)$.

Важи да је

$$[0, 999] = [0, 99] \cup [100, 199] \cup \ldots \cup [900, 999],$$

па је

$$f_0(999) = f(0, 99) + f(100, 199) + \ldots + f(900, 999).$$

Важи да је $f(0, 99) = f(100, 199) = f(200, 299) = f(400, 499) = \ldots = f(900, 999) = f_0(99)$, а да је $f(300, 399) = 0$. Стога је

$$f_0(999) = 8 \cdot f_0(99) + f_0(99) = 9 \cdot f_0(99).$$

Слично, важи да је

$$f_0(99) = 9 \cdot f_0(9),$$

док је $f_0(9) = 9$ (јер у интервалу $[0, 9]$ који има 10 елемената, једино елемент 3 садржи цифру 3).

Важи и да је

$$[0, 251] = [0, 99] \cup [100, 199] \cup [200, 251],$$

па је

$$f_0(251) = 2 \cdot f_0(99) + f_0(51).$$

Пошто је

$$[0, 51] = [0, 9] \cup [10, 19] \cup [20, 29] \cup [30, 39] \cup [40, 49] \cup [50, 51],$$

важи да је

$$f_0(51) = 4 \cdot f_0(9) + f_0(1).$$

Важи и да је $f_0(1) = 2$ јер су оба елемента интервала $[0, 1]$ такви да не садрже цифру 3.

Дакле, $f_0(4251) = 3 \cdot f_0(999) + f_0(251) = 3 \cdot (9 \cdot f_0(99)) + 2 \cdot f_0(99) + f_0(51) = 3 \cdot (9 \cdot (9 \cdot f_0(9))) + 2 \cdot (9 \cdot f_0(9)) + 4 \cdot f_0(9) + f_0(1) = 3 \cdot 9 \cdot 9 \cdot 9 + 2 \cdot 9 \cdot 9 + 4 \cdot 9 + 2 = 2387$.

Функцију f_0 је могуће рекурзивно дефинисати. Ако се број n разлаже на почетну цифру c и суфикс n' тада се $f_0(n)$ може израчунати на следећи начин, у зависности од цифре c (претпостављамо да број $9 \dots 9$ има онолико деветки колико цифара има број n').

- Ако је $c < 3$ тада је $f_0(n) = c \cdot f_0(9 \dots 9) + f_0(n')$,
- Ако је $c = 3$ тада је $f_0(n) = c \cdot f_0(9 \dots 9)$,
- Ако је $c > 3$ тада је $f_0(n) = (c - 1) \cdot f_0(9 \dots 9) + f_0(n')$.

Излаз из рекурзије може бити и само случај $f_0(0) = 1$ (0 је једини број у интервалу $[0, 0]$ и он не садржи цифру 3).

Проблем са имплементацијом овакве рекурзивне функције је то што се цифре одвајају слева надесно, што је компликованије него здесна налево, када број n лако разлажемо на $n \operatorname{div} 10$ и $n \bmod 10$. Стога пре уласка у рекурзију можемо обрнути цифре броја. Такође, за дати број n потребно је одредити одговарајући број који се састоји само од деветки. То једноставно можемо решити тако што конструишемо број који се од броја n добија заменом свих цифара цифром 9 и ако тај број прослеђујемо као други параметар рекурзије (тај број у сваком позиву садржи само деветке и то онолико деветки колико цифара има број n).

Приметимо да се од једног рекурзивног позива за број са k цифара најчешће добијају два рекурзивна позива за бројеве са $k - 1$ цифара, што указује на то да је сложеност експоненцијална (за основу 2), што је допустиво, јер је број цифара мали. Ипак, с обзиром на то да се исти рекурзивни позиви преклапају (пре свега они облика $f_0(9 \dots 9)$), имплементација се може убрзати динамичким програмирањем или тако што се примети да је да је $f_0(\underbrace{9 \dots 9}_{k}) = 9^k$, па би се ови рекурзивни позиви могли специјализовати.

```
int f(int n, int d) {
  if (n == 0)
    return 1;
  int c = n % 10;
  if (c < 3)
    return c * f(d / 10, d / 10) + f(n / 10, d / 10);
  else if (c == 3)
    return c * f(d / 10, d / 10);
  else
    return (c - 1) * f(d / 10, d / 10) + f(n / 10, d / 10);
}

int f(int n) {
  int nObratno = 0;
  int devetke = 0;
  do {
    nObratno = nObratno * 10 + n % 10;
    devetke = devetke * 10 + 9;
    n /= 10;
  } while (n > 0);
  return f(nObratno, devetke);
}
```

Уместо рекурзије која ради одозго наниже, решење можемо синтетизовати одоздо навише.

Обрађиваћемо једну по једну цифру броја n, здесна налево и мало по мало ћемо проширивати обрађени суфикс n' броја n. Уведимо променљиву, нпр. br, која током итерације чува вредности $f_0(n')$ за суфиксе n' броја n које током итерације обрађујемо, и променљиву, нпр. t, која чува вредности бројева $f_0(9 \dots 9) = 9^k$, за бројеве $9 \dots 9$ који имају k цифара 9, колико укупно цифара има у тренуном суфиксу n'.

Променљиве t и br иницијализујемо на 1 (претпостављамо да је пре петље обрађен празан суфикс који одговара броју 0 и да је $9^0 = 1$). Затим у петљи обрађујемо цифру по цифру броја n, кренувши од цифре јединица. Алгоритам издвајања цифара описан је у задатку Број и збир цифара броја. Променљиву br ажурирамо на следећи начин, у зависности од текуће цифре c.

- Ако је $c < 3$ тада је $br = c \cdot t + br$,
- Ако је $c = 3$ тада је $br = c \cdot t$,
- Ако је $c > 3$ тада је $br = (c - 1) \cdot t + br$.

Променљиву t ажурирамо на вредност $9 \cdot t$.

Размотримо поново пример израчунавања $f_0(4251)$ и применимо претходно описани алгоритам на број 4251.

Претпоставимо да на почетку променљиве t и br имају вредност 1.

Прво обрађујемо последњу (прву с десна) цифру броја n, а то је цифра 1. Пошто је она мања од 3, ажурирамо br на вредност $c \cdot t + br = 1 \cdot 1 + 1 = 2$, а вредност t на вредност $9 \cdot t = 9$. Након овога, променљива br има вредност $f_0(1)$, а променљива t има вредност $f_0(9)$.

Наредна цифра је 5 и пошто је она већа од 3, ажурирамо вредност br на $(c - 1) \cdot t + br = 4 \cdot 9 + 2 = 38$, а вредност t на вредност $9 \cdot t = 81$. Након овога, променљива br има вредност $f_0(51)$, а променљива t има вредност $f_0(99)$.

Наредна цифра је 2 и пошто је она мања од 3, ажурирамо вредност br на $c \cdot t + br = 2 \cdot 81 + 38 = 200$, а вредност t на вредност $9 \cdot t = 9 \cdot 81 = 729$. Након овога, променљива br има вредност $f_0(251)$, а променљива t има вредност $f_0(999)$.

На крају, почетна цифра је 4 и пошто је она већа од 4, ажурирамо вредност br на $(c-1) \cdot t + br = 3 \cdot 729 + 200 = 2387$. Вредност t се ажурира на $9 \cdot 729 = 6561$, али се та вредност даље не користи. Након овога, променљива br има вредност $f_0(4251)$, а променљива t има вредност $f_0(9999)$.

```
int t = 1, br = 1;
while (n > 0) {
  int c = n % 10;
  if (c < 3)
    br = c*t + br;
  else if (c == 3)
    br = c*t;
  else
    br = (c-1)*t + br;
  t = 9*t;
  n /= 10;
}
cout << br << endl;
```

Задатак: Дужина сажете форме ниске

Сажете форме ниски се граде од малих слова енглеског алфабета и цифара од 2 до 9. Свака цифра у сажетој форми означава да се ниска одређена префиксом сажете форме пре те цифре понавља онолико пута колика је вредност те цифре. На пример, сажета форма `a2b3` означава ниску `aabaabaab` (а не `aabbb`), јер цифра `2` иза `a` означава да се `a` понавља два пута, па `a2b` представља ниску `aab`, док цифра `3` иза `a2b` означава да се ниска одређена сажетом формом `a2b`, а то је `aab`, понавља три пута. Напиши програм који одређује дужину ниске задате неком сажетом формом.

Улаз: Са стандардног улаза се учитава сажета форма ниске која је дужине мање од 10^{18}.

Излаз: На стандардни излаз исписати дужину сажете форме.

Пример 1

Улаз	*Излаз*
`a2bb3c`	`13`

Пример 2

Улаз	*Излаз*
`a2b32c`	`19`

Решење

Обрада цифара у сажетој форми

Рекурзивна варијанта

Сажета форма s је састављена или само од малих слова (у ком случају је тривијално одредити њену дужину $|s|$) или у себи садржи неку цифру. Ако садржи цифру c, она је облика s_1cs_2, где можемо претпоставити да је c последња цифра, тј. да је s_1 сажета форма, а да је s_2 обична ниска (која не садржи цифре). Тада је $|s| = c \cdot |s_1| + |s_2|$. Дужину сажете форме $|s_1|$ одређујемо рекурзивно, док дужину ниске $|s_2|$ одређујемо тривијално, јер она не садржи карактере. Овим је у потпуности дефинисан рекурзивни поступак за израчунавање дужине сажете форме.

Прикажимо израчунавање дужине ниске $ab2c3d$.

$|ab2c3d| = 3|ab2c| + |d| = 3(2|ab| + |c|) + 1 = 3(2 \cdot 2 + 1) + 1 = 16$

```
#include <iostream>
#include <string>

using namespace std;

typedef long long ll;

ll duzina(const string& s) {
  const string digits = ”0123456789”;
  size_t p = s.find_last_of(digits);
  if (p == string::npos)
    return s.length();
  else
    return duzina(s.substr(0, p)) * (s[p] - '0') + (s.length() - p - 1);
}

int main() {
  string s;
  cin >> s;
  cout << duzina(s) << endl;
  return 0;
}
```

Итеративна варијанта

Дужину можемо израчунати и итеративним поступком, анализирајући једну по једну цифру у сажетом запису. Претпоставићемо да је r дужина обрађеног дела сажете форме s, а да је pp број карактера тог дела сажете форме (обе иницијализујемо на нулу). Све док у преосталом делу сажете форме постоје цифре, радимо следеће. Одређујемо позицију p наредне цифре (то је позиција прве цифре у делу сажете форме иза позиције pp).

- Ако таква позиција постоји, то значи да је сажета форма закључно са позицијом p облика s_1s_2c, где је s_1 сажета форма која садржи pp карактера и представља ниску дужине r, док се c налази на позицији p. Дужина ниске s_2 је $p - pp$, па је дужина ниске представљене сажетом формом s_1s_2c једнака $(r + (p - pp)) \cdot s$ – променљиву r ажурирамо баш на ту вредност.
- На крају, када установимо да иза позиције pp не постоји више цифара знамо да је сажета форма облика s_1s_2, где је ниска представљена формом s_1 дужине r, док је број карактера дела s_2, па и дужина ниске која је њиме представљена једнака разлици између дужине целе сажете форме и позиције pp. Зато се на крају r увећава тачно за ту разлику.

Прикажимо кретање променљивих током обраде сажете форме $ab2c3d$.

```
pp     r     p     obradjen deo forme s
0      0     2     -
3      4     4     ab2
5     15     -     ab2c3
      16           ab2c3d
```

```
#include <iostream>
#include <string>

using namespace std;

typedef long long ll;

ll duzina(const string& s) {
  const string digits = ”0123456789”;
  size_t pp = 0;
  size_t p = s.find_first_of(digits);
  ll rez = 0;
  while (p != string::npos) {
    rez += p - pp;
    rez *= s[p] - '0';
    pp = p+1;
    p = s.find_first_of(digits, p+1);
  }
  rez += s.length() - pp;
  return rez;
}

int main() {
  string s;
  cin >> s;
  cout << duzina(s) << endl;
  return 0;
}
```

Анализа појединачних карактера у сажетој форми

Рекурзивна варијанта

Рекурзивну функцију је могуће организовати и на основу анализе последњег карактера сажете форме.

- Ако је сажета форма празна дужина ниске коју представља је нула.
- Ако је последњи карактер сажете форме слово, онда је дужина ниске коју представља за један већа од дужине ниске коју представља сажета форма без тог последњег слова.
- Ако је последњи карактер сажете форме цифра, онда је дужина ниске коју представља једнака производу те цифре и дужине ниске коју представља сажета форма без те последње цифре.

Прикажимо поступак израчунавања дужине сажете форме $ab2c3d$.

$$\begin{aligned} |ab2c3d| &= |ab2c3| + 1 = |ab2c| \cdot 3 + 1 = (|ab2| + 1) \cdot 3 + 1 = (|ab| \cdot 2 + 1) \cdot 3 + 1 \\ &= ((|a| + 1) \cdot 2 + 1) \cdot 3 + 1 = (((|\varepsilon| + 1) + 1) \cdot 2 + 1) \cdot 3 + 1 \\ &= (((0 + 1) + 1) \cdot 2 + 1) \cdot 3 + 1 = 16 \end{aligned}$$

```
#include <iostream>
#include <string>

using namespace std;

// duzina prefiksa duzine n sazete forme s
long long duzina(const string& s, int n) {
  if (n == 0)
    return 0;
  if (!isdigit(s[n-1]))
    return duzina(s, n-1) + 1;
```

```
  else
    return duzina(s, n-1) * (s[n-1] - '0');
}

// duzina sazete forme s
long long duzina(const string& s) {
  return duzina(s, s.length());
}

int main() {
  string s;
  cin >> s;
  cout << duzina(s) << endl;
  return 0;
}
```

Итеративна варијанта

Дужину можемо израчунати и итеративним поступком, анализирајући један по један карактер слева надесно. У сваком тренутку број r представља дужину ниске представљене до тада обрађеним карактерима сажете форме s. На почетку није обрађен ни један карактер, па се r иницијализује на нулу. Ако је текући карактер слово, r се увећава за 1, а ако је цифра, тада се r множи том цифром.

Прикажимо кретање промењљивих током обраде сажете форме $ab2c3d$.

```
r    c   obradjen deo forme s
0    a    -
1    b    a
2    2    ab
4    c    ab2
5    3    ab2c
15   d    ab2c3
16        ab2c3d
```

```
#include <iostream>
#include <string>

using namespace std;

typedef long long ll;

ll duzina(const string& s) {
  // duzina prefiksa sazete forme s koji se sastoji od do sada
  // obradjenih karaktera niske s
  ll rez = 0;
  // obradjujemo karakter po karakter i azuriramo duzinu obradjenog dela
  // sazete forme s
  for (char c : s)
    if (isdigit(c))
      rez *= c - '0';
    else
      rez++;
  return rez;
}

int main() {
  string s;
  cin >> s;
  cout << duzina(s) << endl;
  return 0;
}
```

Задатак: Слово у сажетој форми ниске

Сажете форме ниски се граде од малих слова енглеског алфабета и цифара од 2 до 9. Свака цифра у сажетој форми означава да се ниска одређена префиксом сажете форме пре те цифре понавља онолико пута колика је вредност те цифре. На пример, сажета форма `a2b3` означава ниску `aabaabaab` (а не `aabbb`), јер цифра `2` иза `a` означава да се `a` понавља два пута, па `a2b` представља ниску `aab`, док цифра `3` иза `a2b` означава да се ниска одређена сажетом формом `a2b`, а то је `aab`, понавља три пута. Напиши програм који одређује елемент на позицији k унутар ниске задате неком сажетом формом.

Улаз: Са стандардног улаза се учитава сажета форма ниске која је дужине мање од 10^{18}, а затим и број k који је већи или једнак од 0, а строго мањи од дужине ниске.

Излаз: На стандардни излаз исписати тражено слово.

Пример 1

Улаз	*Излаз*
`zdravo2svima3` `30`	`v`

Пример 2

Улаз	*Излаз*
`a2b3ca3d2f` `40`	`a`

Решење

Рекурзивно решење

Сажета форма ниске је или састављена само од малих слова или је облика $s_1 c s_2$, где је c цифра, а s_2 је ниска састављена само од малих слова.

- Ако се сажета форма се састоји само од малих слова, она је једнака ниски коју представља и њено слово на позицији k се тривијално очитава.
- Ако је сажета форма облика $s_1 c s_2$, тада она представља ниску која се састоји од c копија ниске представљене сажетом формом s_1 на које је надовезана ниска s_2. Ако је $d = |s_1|$ дужина ниске представљене сажетом формом s_1, разликоваћемо случај када је $k < c \cdot d$ и када је $k \geq c \cdot d$. Дужину сажете форме можемо израчунати на неки од начина описаних као у задатку Дужина сажете форме ниске.
  - Ако је $k < c \cdot d$, тада треба израчунати слово на позицији k у c поновљених копија ниске представљене сажетом формом s_1 (њихова укупна дужина је $c \cdot d$). Пошто се слова циклично понављају са периодом d, потребно је одредити слово на позицији $k \bmod d$ у ниски представљеној сажетом формом s_1, што се може урадити рекурзивним позивом.
  - Ако је $k \geq c \cdot d$, тада слово припада ниски s_2 и налази се у њој на позицији $k - c \cdot d$.

Размотримо пример проналажења карактера на позицији 40 у ниски одређеној сажетом формом $a2b3ca3d2f$.

- Сажета форма $s = a2b3ca3d2f$ се разлаже на $s_1 = a2b3ca3d$, $c = 2$ и $s_2 = f$. Дужина $d = |s_1| = 34$, па је $40 = k < cd = 68$. Зато одређујемо карактер на позицији $40 \bmod 34 = 6$ у ниски одређеној сажетом формом $a2b3ca3d$.
- Сажета форма $s = a2b3ca3d$ се разлаже на $s_1 = a2b3ca$, $c = 3$ и $s_2 = d$. Дужина $d = |s_1| = 11$, па је $6 = k < cd = 33$. Зато одређујемо карактер на позицији $6 \bmod 11 = 6$ у ниски одређеној сажетом формом $a2b3ca$.
- Сажета форма $s = a2b3ca$ се разлаже на $s_1 = a2b$, $c = 3$ и $s_2 = ca$. Дужина $d = |s_1| = 3$, па је $6 = k < cd = 9$. Зато одређујемо карактер на позицији $6 \bmod 3 = 0$ у ниски одређеној сажетом формом $a2b$.
- Сажета форма $a2b$ се разлаже на $s_1 = a$, $c = 2$ и $s_2 = b$. Дужина $d = |s_1| = 1$, па је $0 = k < cd = 2$. Зато одређујемо карактер на позицији $0 \bmod 1 = 0$ у ниски одређеној сажетом формом a.
- Ниска a не садржи цифре, па читамо њен карактер на позицији 0 и то је a.

Директна имплементација претходног алгоритма која израчунава дужину подниске тако што сваки пут позива функцију која израчунава дужину сажете форме није најефикаснија, јер долази до понављања идентичних израчунавања. Ово би се се могло оптимизовати тако што би се применила мемоизација и тако што би се дужине свих префикса сажете форме упамтиле у низ. Уместо експлицитног издвајања префикса (функцијом за издвајање подниске), могли бисмо чувати стално оригиналну ниску и позицију која одређује дужину префикса. С обзиром на то да је сажета форма (за разлику од ниске која је њоме представљена) обично веома кратка, ове оптимизације нису неопходне.

```
#include <iostream>
#include <string>

using namespace std;

typedef long long ll;

const string digits = "0123456789";

ll duzina(const string& s) {
  size_t p = s.find_last_of(digits);
  if (p == string::npos)
    return s.length();
  else
    return duzina(s.substr(0, p)) * (s[p] - '0') + (s.length() - p - 1);
}

char kToSlovo(const string& s, ll k) {
  size_t p = s.find_last_of(digits);
  if (p == string::npos)
    return s[k];
  ll d = duzina(s.substr(0, p));
  int m = s[p] - '0';
  if (k >= m*d)
    return s.substr(p+1)[k-m*d];
  else
    return kToSlovo(s.substr(0, p), k % d);
}

int main() {
  string s;
  cin >> s;
  ll k;
  cin >> k;
  cout << kToSlovo(s, k) << endl;
  return 0;
}
```

Итеративно решење

Задатак можемо решити и итеративно. У првој фази можемо поделити сажету форму на делове који садрже подниске састављене само од слова и цифре које одређују број понављања. Ако се сажета форма не завршава цифром, униформности ради можемо је допунити цифром 1.

На пример, сажету форму $a2b3ca3d2f$ разлажемо на $(a, 2), (b, 3), (ca, 3), (d, 2), (f, 1)$.

Након тога израчунавамо дужине свих ниски одређених префиксима сажете форме. Ако смо сажету форму разложили на низ парова $(s_0, c_0), (s_1, c_1), \ldots, (s_m, c_m)$, тада рачунамо редом дужине ниски одређених префиксима $d_0 = |s_0|$, $d_1 = |s_0c_0s_1|$, $d_2 = |s_0c_0s_1c_1s_2|$ итд., закључно са дужином $d_m = |s_0c_0s_1c_1 \ldots c_{m-1}s_m|$. Дужина првог префикса је $d_0 = |s_0|$, док се дужина наредних префикса израчунава као $d_i = d_{i-1} \cdot c_{i-1} + |s_i|$, за $i > 0$.

Дужине ниски одређене префиксима сажете форме $a2b3ca3d$ су одређене на следећи начин.

- $d_0 = |a| = 1$
- $d_1 = |a2b| = 1 \cdot 2 + 1 = 3$
- $d_2 = |a2b3ca| = 3 \cdot 3 + 2 = 11$
- $d_3 = |a2b3ca3d| = 11 \cdot 3 + 1 = 34$
- $d_4 = |a2b3ca3d2f| = 34 \cdot 2 + 1 = 69$

На крају анализу крећемо од краја тј. од пара (s_m, c_m), па уназад до пара (s_0, c_0). У сваком кораку одређујемо

елемент на позицији k у ниски одређеној префиксом $s_0c_0 \dots c_{i-1}s_ic_is_{i+1}$ (када је $i = m$, тада је s_{i+1} празна ниска).

- Ако је $k \geq c_i \cdot d_i = c_i|s_0c_0 \dots c_is_i|$, тада тражено слово припада ниски s_{i+1} на позицији $k - d_ic_i$. Ово се по условима задатка не може догодити када је $i = m$.
- У супротном се слово налази у ниски одређеној помоћу c_i понављања ниске одређене префиксом $s_0c_0 \dots c_{i-1}s_i$, чија је дужина d_i. То се слово налази на позицији $k \bmod d_i$ у ниски одређеној префиксом $s_0c_0 \dots c_{i-1}s_i$, па стога k мењамо његовим остатком при дељењу са d_i и прелазимо на наредну итерацију.

Ако се петља заврши а да нисмо већ одредили тражено слово, знамо да се карактер налази у ниски s_0 и то на позицији k.

Прикажимо извршавање алгоритма на примеру проналажења карактера на позицији $k = 40$ унутар ниске одређене сажетом формом $a2b3ca3d2f$.

- Крећемо од вредности $i = m = 4$. Важи да је $k = 40 < c_4d_4 = 1 \cdot 69 = 69$, па одређујемо остатак при дељењу 40 са 69, али тиме k остаје 40.
- Прелазимо на $i = 3$. Важи да је $k = 40 < c_3d_3 = 2 \cdot 34 = 68$, па одређујемо остатак при дељењу 40 са 34 и мењамо k на 6.
- Прелазимо на $i = 2$. Важи да је $k = 6 < c_2d_2 = 3 \cdot 11 = 33$, па одређујемо остатак при дељењу 6 са 11, али тиме k остаје 6.
- Прелазимо на $i = 1$. Важи да је $k = 6 < c_1d_1 = 3 \cdot 3 = 9$, па одређујемо остатак при дељењу 6 са 3 и мењамо k на 0.
- Прелазимо на $i = 0$. Важи да је $k = 0 < c_0d_0 = 2 \cdot 1 = 2$, па одређујемо остатак при дељењу 0 са 1, али тиме k остаје 0.

Пошто је петља завршена, враћамо карактер на позицији 0 ниске s_0, а то је a.

```
#include <iostream>
#include <vector>
#include <string>
#include <utility>

using namespace std;

typedef long long ll;

const string digits = "0123456789";

vector<pair<string, int>> podeli(const string& s) {
  vector<pair<string, int>> rez;
  size_t pp = 0, p = s.find_first_of(digits);
  while (p != string::npos) {
    rez.emplace_back(s.substr(pp, p - pp), s[p] - '0');
    pp = p + 1;
    p = s.find_first_of(digits, p + 1);
  }
  if (pp < s.length())
    rez.emplace_back(s.substr(pp), 1);
  return rez;
}

char kToSlovo(const string& s, ll k) {
  auto delovi = podeli(s);

  vector<ll> duzine(delovi.size());
  duzine[0] = delovi[0].first.length();
  for (int i = 1; i < delovi.size(); i++)
```

```
    duzine[i] = duzine[i-1] * delovi[i-1].second + delovi[i].first.length();

  for (int i = delovi.size() - 1; i >= 0; i--) {
    if (k >= duzine[i] * delovi[i].second)
      return delovi[i+1].first[k - duzine[i]*delovi[i].second];
    k %= duzine[i];
  }
  return delovi[0].first[k];
}

int main() {
  string s;
  cin >> s;
  ll k;
  cin >> k;
  cout << kToSlovo(s, k) << endl;
  return 0;
}
```

Задатак: Ханојске куле

Од три дата штапа, на једном је n дискова различитих величина, а остала два су празна. Дискови на првом штапу су поређани по величини, то јест тако да је диск величине n на дну, на њему је диск величине $n-1$ итд. све до диска величине 1, који је на врху.

Потребно је дискове преместити са првог на трећи штап користећи што мање премештања. При томе треба премештати дискове један по један и стављати само мањи диск на већи, а никако обрнуто (није дозвољено стављати већи диск преко мањег).

На пример, ако је $n = 3$, редослед премештања треба да буде:

```
диск величине 1 са штапа 1 на штап 3
диск величине 2 са штапа 1 на штап 2
диск величине 1 са штапа 3 на штап 2
диск величине 3 са штапа 1 на штап 3
диск величине 1 са штапа 2 на штап 1
диск величине 2 са штапа 2 на штап 3
диск величине 1 са штапа 1 на штап 3
```

Напиши програм који за дати број дискова n исписује редослед премештања.

Улаз: Са стандардног улаза се учитава број штапова n ($1 \leq n \leq 10$).

Излаз: На стандардни излаз за свако премештање једног диска исписати по један ред, у коме се наводи редни број штапа са чијег врха се диск премешта и редни број штапа на чији врх се диск премешта, раздвојене једним размаком.

Пример 1

Улаз	*Излаз*
3	1 3
	1 2
	3 2
	1 3
	2 1
	2 3
	1 3

Пример 2

Улаз	*Излаз*
2	1 2
	1 3
	2 3

Решење

Претпоставимо да умемо да решимо задатак за мање од n дискова. Тада решење за n дискова можемо да добијемо на следећи начин:

- Познатим поступком преместимо $n-1$ диск са штапа 1 на штап 2 (користећи штап 3 као помоћни)

- Једним премештањем пребацимо највећи диск са штапа 1 на штап 3
- Познатим поступком преместимо $n-1$ диск са штапа 2 на штап 3 (користећи штап 1 као помоћни)

Први и трећи корак се своде на рекурзивне позиве исте функције, а други корак на једну наредбу исписивања.

Пошто при рекурзивним позивима за мањи број дискова штапови мењају улоге, функција која решава задатак треба да има четири параметра: број дискова, редни број штапа са којег се дискови премештају, редни број штапа који се користи као помоћни и редни број штапа на који се дискови премештају.

```
#include <iostream>

using namespace std;

void Hanoj(int n, int sa, int pom, int na) {
    if (n > 0) {
        Hanoj(n-1, sa, na, pom);
        cout << sa << " " << na << endl;
        Hanoj(n-1, pom, sa, na);
    }
}

int main() {
  unsigned n;
  cin >> n;
  Hanoj(n, 1, 2, 3);
  return 0;
}
```

Задатак: Функција пар-непар

Функција f је задата са: $f(1) = 1, f(2n) = f(n), f(2n+1) = f(n) + f(n+1)$.

Напиши програм који за дато n исписује $f(n)$.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10\,000\,000\,000\,000\,000\,000 = 10^{19}$).

Излаз: На стандардни излаз исписати тражену вредност функције.

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
3	2	64	1	10000000000000000000	59010673

Решење

Рекурзивно решење произилази директно из дефиниције функције.

Мана овог решења је извесна спорост. Пошто се из рекурзивне функције излази само за $n = 1$, резултат $f(n)$ показује колико пута је позвано $f(1)$, тј. колико листова постоји у дрвету свих позива.

```
#include <iostream>

using namespace std;

int f(unsigned long long  n) {
    if (n == 1) return 1;
    unsigned long long n2 = n / 2;
    if (n2 + n2 == n) return f(n2);
    return f(n2) + f(n2 + 1);
}

int main() {
  unsigned long long n;
  cin >> n;
  cout << f(n) << endl;
```

```
  return 0;
}
```

Да бисмо боље разумели шта је потребно урадити да се добије итеративно решење, израчунајмо ручно нпр. $f(21)$

```
f(21) = f(10) + f(11)
      = f(5) + [f(5) + f(6)] = 2 f(5) + f(6)
      = 2 [f(2) + f(3)] + f(3) = 2 f(2) + 3 f(3)
      = 2 f(1) + 3 [f(1) + f(2)] = 5 f(1) + 3 f(2)
      = 5 + 3 f(1)
      = 5 + 3 = 8
```

Видимо да се у свакој итерацији добија израз облика $a \cdot f(m) + b \cdot f(m+1)$, при чему се m у свакој итерацији приближно преполови. Почетни израз $f(n)$ је такође овог облика, јер $f(n) = 1 \cdot f(n) + 0 \cdot f(n+1)$. Начин трансформисања израза зависи од парности броја m у текућој итерацији. Размотримо оба случаја.

За парно m, тј. $m = 2k$ имамо:

```
a f(2k) + b f(2k+1) =
a f(k) + b [f(k) + f(k+1)] = (a+b) f(k) + b f(k+1)
```

а за непарно m, тј. $m = 2k + 1$ важи:

```
a f(2k+1) + b f(2k+2) =
a [f(k) + f(k+1)] + b f(k+1) = a f(k) + (a+b) f(k+1)
```

Са итерацијама завршавамо када стигнемо до $k = 1$, јер је тада резултат познат и једнак $a + b$.

На основу овог разматрања можемо да напишемо програм.

```
#include <iostream>

using namespace std;

int f(unsigned long long  n) {
    int a = 1, b = 0;
    while (n > 1) {
        unsigned long long n2 = n / 2;
        if (n2 + n2 == n) a += b;
        else b += a;
        n = n2;
    }
    return a + b;
}

int main() {
  unsigned long long n;
  cin >> n;
  cout << f(n) << endl;
  return 0;
}
```

3.4 Претрага у дубину

Задатак: Број белих области

Написати програм којим се за црно-белу матрицу (0 – бела, 1 – црна боја) одређује број белих области. Белу област чине повезана бела поља. Два бела поља су повезана ако су она почетно и крајње поље неког низа белих поља у коме узастопна поља имају заједничку страницу.

Улаз: У првој линији стандардног улаза се учитава број редова матрице n ($2 \leq n \leq 20$), у другој број колона m ($2 \leq m \leq 20$). У следећих n редова учитава се по m бројева чија је вредност 0 или 1.

Излаз: Број белих области.

Пример

Улаз	*Излаз*
5 6 1 1 1 1 1 0 1 1 0 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0	3

Решење

Основу решења чини рекурзивна функција која детектује и обележава белу област почевши од једног њеног поља. Функција се позива за поље у матрици које је тренутно обојено бело (у матрици на том пољу пише 0), обележава га (уписује неку другу вредност, на пример, -1), и затим се рекурзивно позива за све његове суседе беле боје (при том треба водити рачуна о томе да суседни случајно не испадну из матрице).

У главном програму обилазимо сва поља матрице (коришћењем угнежђених петљи) и за свако бело поље (оно је део неке нове области) позивамо рекурзивну функцију која ће детектовати и обележити сва поља те области и увећавамо бројач обележених области. Када се обиђе цела матрица, све беле области су обележене, па можемо исписати њихов број.

```
#include <iostream>
using namespace std;

// maksimalne dimenzije matrice
const int N = 20, M = 20;

// oznake polja u matrici
const int BELA = 0, CRNA = 1, OBELEZENO = -1;

// provera da li se polje (x, y) nalazi u matrici dimenzije mxn
bool UMatrici(int x, int y, int m, int n) {
  return x >= 0 && x < m && y >= 0 && y < n;
}

// obilazi se belo, neobelezeno polje sa koordinatama (x, y)
void Obelezi(int x, int y, int a[N][M], int m, int n) {
  // obelezavamo polje
  a[x][y] = OBELEZENO;
  // obilazimo sve susede
  int dx[] = { -1, 0, 1, 0 };
  int dy[] = { 0, 1, 0, -1 };
  for (int i = 0; i < 4; i++) {
    int xx = x + dx[i], yy = y + dy[i];
    // ako je susedno polje belo (i neobelezeno), obelezavamo ga
    if (UMatrici(xx, yy, m, n) && a[xx][yy] == BELA)
      Obelezi(xx, yy, a, m, n);
  }
}

int main() {
  // ucitavamo matricu
  int m, n;
  cin >> m >> n;
  int a[M][N];
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      cin >> a[i][j];
```

```
  int oblast = 0;   // broj obelezenih oblasti
  // obilazimo sva polja u matrici
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      if (a[i][j] == BELA) {
        Obelezi(i, j, a, m, n);
        oblast++;
      }

  cout << oblast << endl;
}
```

Претрагу у дубину је могуће имплементирати и уз помоћ стека.

```
#include <iostream>
#include <stack>
#include <utility>
using namespace std;

// maksimalne dimenzije matrice
const int N = 20, M = 20;

// oznake polja u matrici
const int BELA = 0, CRNA = 1, OBELEZENO = -1;

// provera da li se polje (x, y) nalazi u matrici dimenzije mxn
bool UMatrici(int x, int y, int m, int n) {
  return x >= 0 && x < m && y >= 0 && y < n;
}

// obilazi se belo, neobelezeno polje sa koordinatama (x, y)
void Obelezi(int x0, int y0, int a[N][M], int m, int n) {
  stack<pair<int, int>> obeleziti;
  obeleziti.emplace(x0, y0);
  while (!obeleziti.empty()) {
    auto p = obeleziti.top(); obeleziti.pop();
    int x = p.first, y = p.second;
    // obelezavamo polje
    a[x][y] = OBELEZENO;
    // obilazimo sve susede
    int dx[] = { -1, 0, 1, 0 };
    int dy[] = { 0, 1, 0, -1 };
    for (int i = 0; i < 4; i++) {
      int xx = x + dx[i], yy = y + dy[i];
      // ako je susedno polje belo (i neobelezeno), obelezavamo ga
      if (UMatrici(xx, yy, m, n) && a[xx][yy] == BELA)
        obeleziti.emplace(xx, yy);
    }
  }
}

int main() {
  // ucitavamo matricu
  int m, n;
  cin >> m >> n;
  int a[M][N];
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
```

```
      cin >> a[i][j];

  int oblast = 0;   // broj obelezenih oblasti
  // obilazimo sva polja u matrici
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      if (a[i][j] == BELA) {
        Obelezi(i, j, a, m, n);
        oblast++;
      }

  cout << oblast << endl;
}
```

Види другачија решења овог задатка.

Задатак: Највећа заражена област

Ако је матрицом 0 и 1 дата мапа која изражава зараженост биљног света у некој области, где 0 представља незаражено поље области, а 1 заражено, одредите број поља максималне заражене области. Два заражена поља припадају истој области ако им се странице наслањају једна на другу.

Улаз: У првој линији стандардног улаза се учитава број редова матрице n ($2 \leq n \leq 20$), у другој број колона m ($2 \leq m \leq 20$). У следећих n редова учитава се по m бројева чија је вредност 0 или 1.

Излаз: Број поља највеће заражене области (0 ако нема заражених поља).

Пример

Улаз	*Излаз*
5 13 1 1 1 0 1 0 1 1 1 0 0 1 0 1 1 1 0 0 0 1 1 0 1 1 0 1 0 0 1 0 0 1 0 1 1 0 1 1 1 1 0 1 0 1 0 0 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 0 1 1 1	13

Задатак: Minesweeper отварање

Напиши програм који приказује садржај поља у игрици Minewsweeper (ако не знаш како изгледа, потражи на интернету) након отварања једног поља.

Улаз: Са стандардног улаза се учитава матрица димензије 10 пута 10 која на пољима где су бомбе садржи јединице, а на слободним пољима садржи нуле. У наредном реду се учитавају координате поља које се отвара (број врсте и број колоне између 0 и 9, раздвојени једним размаком).

Излаз: На стандардни излаз исписати стање које се кориснику приказује након отварања тог поља. Ако је на пољу бомба, исписати `boom`. У супротном приказати матрицу димензије 10 пута 10 тако да се на неотвореним пољима приказује `x` на празним пољима (пољима која су отворена и немају бомби у околини) приказује `.`, а на отвореним пољима која имају бомбе у околини приказује број бомби у околини.

Пример

Улаз	*Излаз*
0100010100	xxxxxxxxxx
0100111000	xxxxxxxxxx
1000000000	xx21233xxx
1100000110	xx1...1xxx
1000000000	xx2...2xxx
1100000100	xx421.1xxx
0111000011	xxxx113xxx
1100001110	xxxxxxxxxx
0011100000	xxxxxxxxxx
1110000001	xxxxxxxxxx
5 5	

Задатак: P-Q коњи

Написати програм којим се одређује минимални број (P, Q) - коња којима се могу контролисати (обићи) сва поља шаховске табле димензија nxm.

(P, Q) - коњ је фигура која се у једном скоку премета за P поља по хоризонтали и Q поља по вертикали или Q поља по хоризонтали и P поља по вертикали. Например, $(1, 2)$-коњ је обичан шаховски коњ.

Фигура или група фигура контролишу поље ако могу до њега стићи у нула или више скокова. На пример, за контролу стандардне шаховске табле димензија $8x8$ потребна су два $(1, 1)$-коња (то су фигуре сличне ловцима, али се крећу за по једно поље).

Улаз: У првој линији стандардног улаза се учитава број редова матрице m ($2 \leq m \leq 20$), у другој број колона n ($2 \leq n \leq 20$), у трећој дужина скока P и четвртој дужина скока Q.

Излаз: Природан број који представља број (P, Q) коња.

Пример

Улаз	*Излаз*
8	10
8	
3	
5	

3.5 Рекурзивни спуст

Задатак: Превођење потпуно заграђеног израза у постфиксни облик

Написати програм који исправан инфиксни аритметички израз који има заграде око сваке примене бинарног оператора преводи у постфиксни облик. Једноставности ради претпоставити да су сви операнди једноцифрени бројеви и да се јављају само операције сабирања и множења.

Улаз: Једина линија стандардног улаза садржи исправан, потпуно заграђен израз.

Излаз: На стандардни излаз исписати тражени постфиксни облик.

Пример

Улаз	*Излаз*
(((3*5)+(7+(2*1)))*4)	35*721*++4*

Решење

Чињеница да је израз потпуно заграђен олакшава израчунавање, јер нема потребе да водимо рачуна о приоритету и асоцијативности оператора. Такви изрази се описују наредном, веома једноставном граматиком.

```
<izraz> :: <cifra>
<izraz> :: '(' <izraz> '+' <izraz> ')'
<izraz> :: '(' <izraz> '*' <izraz> ')'
```

Један начин да се приступи решавању проблема је да се примени индуктивно-рекурзивни приступ. Обрада структурираног улаза рекурзивним функцијама се назива *рекурзивни спуст* и детаљно се изучава у курсевима

превођења програмских језика. Дефинишемо рекурзивну функцију чији је задатак да преведе део ниске који представља исправан инфиксни израз. Он може бити или број, када је превођење тривијално јер се он само препише на излаз или израз у заградама. У овом другом случају читамо отворену заграду, затим рекурзивним позивом преводимо први операнд, након тога читамо оператор, затим рекурзивним позивом преводимо други операнд, након тога читамо затворену заграду и исписујемо оператор који смо прочитали (он бива исписан непосредно након превода својих операнада).

Променљива `i` мења своју вредност кроз рекурзивне позиве. Стога ћемо је преносити по референци тако да представља и улазну и излазну величину функције. Задатак функције је да прочита израз који почиње на позицији i, да га преведе у постфиксни облик и да променљиву `i` промени тако да њена нова вредност i' указује на позицију ниске непосредно након израза који је преведен.

```
#include <iostream>
#include <string>

using namespace std;

// Prevodi deo izraza od pozicije i u postfiksni oblik i rezultat
// nadovezuje na nisku postfiks. Po zavrsetku rada funkcije,
// promenljiva i ukazuje na poziciju iza prevedenog izraza.
void prevedi(const string& izraz, int& i, string& postfiks) {
  if (isdigit(izraz[i]))
    postfiks += izraz[i++];
  else {
    // preskačemo otvorenu zagradu
    i++;
    // prevodimo prvi operand
    prevedi(izraz, i, postfiks);
    // pamtimo operator
    char op = izraz[i++];
    // prevodimo drugi operand
    prevedi(izraz, i, postfiks);
    // preskačemo zatvorenu zagradu
    i++;
    // ispisujemo upamćeni operator
    postfiks += op;
  }
}

// prevodi potpuno zagradjen izraz u postfiksni oblik
string prevedi(const string& izraz) {
  string postfiks = "";
  int i = 0;
  prevedi(izraz, i, postfiks);
  return postfiks;
}

int main() {
  string izraz;
  cin >> izraz;
  string postfiks = prevedi(izraz);
  cout << postfiks << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Израз у коме нема заграда

Напиши програм који израчунава вредност аритметичког израза у којем се јављају природни бројеви и између њих оператори `+`, `-` и `*`, али не и заграде. Нпр. `3+4*5-7*2`.

Улаз: Једина линија стандардног улаза садржи описани израз.

Излаз: Стандардни излаз треба да садржи само тражену вредност учитаног израза.

Пример

Улаз	*Излаз*
`3+4*5-7*2`	`9`

Решење

Рекурзивни спуст

Канонско решење овог задатка је да се примени техника рекурзивног спуста. Граматика којом се могу описати ови изрази је следећа

```
E -> E + T
  |  E - T
  |  T
T -> T * num
  | num
```

Ослобађањем од леве рекурзије добијамо граматику:

```
E -> T E'
E' -> + T E'
    | - T E'
    | eps
T -> num T'
T' -> * num T'
    | eps
```

Имплементација је на даље једноставна. Сваком нетерминалу у граматици придружујемо једну функцију. Да бисмо постигли коректно израчунавање и леву асоцијативност (која нам је битна због присуства одузимања), функције које одговарају симболима `E'` и `T'` као аргументе примају вредност израза са своје леве стране.

```
#include <iostream>
#include <string>
#include <cctype>

using namespace std;

enum token {BROJ, PLUS, MINUS, PUTA, EOI};
int _vrednost;
int _poz;
string _ulaz;
token _preduvid;

token lex() {
  if (_poz >= _ulaz.length())
    return EOI;
  char c = _ulaz[_poz++];
  if (c == '+')
    return PLUS;
  if (c == '-')
    return MINUS;
  if (c == '*')
    return PUTA;
  _vrednost = c - '0';
  while (_poz < _ulaz.length() && isdigit(_ulaz[_poz]))
```

```
    _vrednost = 10 * _vrednost + _ulaz[_poz++] - '0';
  return BROJ;
}

int E();
int EP(int);
int T();
int TP(int);

int E() {
  // E -> T E'
  return EP(T());
}

int EP(int x) {
  if (_preduvid == PLUS) {
    // E' -> + T E'
    _preduvid = lex();
    return EP(x + T());
  } else if (_preduvid == MINUS) {
    // E' -> - T E'
    _preduvid = lex();
    return EP(x - T());
  } else {
    // E' -> eps
    return x;
  }
}

int T() {
  // T -> num T'
  int a = _vrednost;
  _preduvid = lex();
  return TP(a);
}

int TP(int x) {
  if (_preduvid == PUTA) {
    // T' -> * num T'
    _preduvid = lex();
    int a = _vrednost;
    _preduvid = lex();
    return TP(x * a);
  } else {
    // T' -> eps
    return x;
  }
}

int vrednost(const string& s) {
  _ulaz = s;
  _poz = 0;
  _preduvid = lex();
  return E();
}

int main() {
  string s;
```

```
  getline(cin, s);
  cout << vrednost(s) << endl;
  return 0;
}
```

Пошто је рекурзија у свим случајевима репна, могуће је потпуно је уклонити.

```
#include <iostream>
#include <string>
#include <cctype>

using namespace std;

enum token {BROJ, PLUS, MINUS, PUTA, EOI};
int _vrednost;
int _poz;
string _ulaz;
token _preduvid;

token lex() {
  if (_poz >= _ulaz.length())
    return EOI;
  char c = _ulaz[_poz++];
  if (c == '+')
    return PLUS;
  if (c == '-')
    return MINUS;
  if (c == '*')
    return PUTA;
  _vrednost = c - '0';
  while (_poz < _ulaz.length() && isdigit(_ulaz[_poz]))
    _vrednost = 10 * _vrednost + _ulaz[_poz++] - '0';
  return BROJ;
}

int T() {
  int x = _vrednost;
  _preduvid = lex();
  while (_preduvid == PUTA) {
    _preduvid = lex();
    x *= _vrednost;
    _preduvid = lex();
  }
  return x;
}

int E() {
  int x = T();
  while (_preduvid == PLUS || _preduvid == MINUS) {
    if (_preduvid == PLUS) {
      _preduvid = lex();
      x += T();
    } else { // if (preduvid == MINUS)
      _preduvid = lex();
      x -= T();
    }
  }
  return x;
}
```

```
int vrednost(const string& s) {
  _ulaz = s;
  _poz = 0;
  _preduvid = lex();
  return E();
}

int main() {
  string s;
  getline(cin, s);
  cout << vrednost(s) << endl;
  return 0;
}
```

Рачунање вредности текућег сабирка

Можемо да памтимо вредност израза без последњег сабирка и посебно вредност последњег (текућег) сабирка. Вредност последњег сабирка иницијализујемо на 1, читамо број и оператор иза њега, вредност последњег сабирка множимо са бројем и ако је прочитани оператор + или - или ако смо стигли до краја ниске, завршили смо управо са израчунавањем тог сабирка, и његову додајемо или одузимамо од резултата у зависности од тога који је био претходни адитивни оператор. Ако смо прочитали оператор + или -, памтимо га. Вредност првог сабирка можемо израчунати засебно и резултат иницијализовати на њега, а можемо на почетку претпоставити да је претходни адитивни оператор био + (иако он не пише испред израза) и први сабирак обрадити као и све остале.

Прикажимо рад овог алгоритма на изразу `8+4*5*2-7*2`.

У старту вредност израза постављамо на нулу, вредност претходног сабирка на један, а претходни оператор на + (еквивалентно, можемо увести променљиву у којој памтимо знак сабирка и можемо је иницијализовати на 1). Након читања броја `8`, множимо претходни сабирак који има иницијалну вредност `1` са њим, добијамо `8` и пошто смо наишли на +, завршили смо са обрадом једог сабирка и вредност текућег резултата која је `0` увећавамо за производ вредности `8` (то је вредност управо обрађеног сабирка) и `1` (то је вредност знака).

Вредност знака опет постављамо на `1` (јер смо наишли на + и у наредном кораку ће опет бити потребно увећање вредности), а вредност текућег сабирка поново враћамо на иницијалну вредност `1`. Читамо затим `4` и множимо текући сабирак са `4` (добијамо вредност `4`), затим читамо `5` и множимо га са `5` (добијамо вредност `20`) и затим читамо `2` и множимо га са `2` (чиме добијамо `40`). Пошто је следећи оператор `-`, завршили смо са обрадо још једног сабирка и резултат увећавамо за вредност текућег сабирка помножену вредношћу знака (добијамо вредност `48`).

Знак постављамо на `-1` (јер ће се наредни сабирак одузимати) и вредност текућег сабирка поново враћамо на иницијалну вредност `1`. Након тога читамо `7` и њиме множимо вредност текућег сабирка (добијамо вредност `7`), затим читамо `2` и њоме множимо вредност текућег сабирка (добијамо вредност `14`) и пошто смо стигли до краја, резултат увећавамо за производ знака и вредности текућег сабирка (тј. на збир додајемо `-14`), чиме добијамо коначну вредност `34`.

```
#include <iostream>
#include <string>

using namespace std;

int vrednost(const string& s) {
  int rezultat = 0;
  int znakTekucegSabirka = 1;
  int tekuciSabirak = 1;
  int tekuciBroj = 0;
  for (int i = 0; i <= s.length(); i++)
    if (i < s.length() && isdigit(s[i]))
      // procitali smo cifru
      // dodajemo je kao poslednju cifru tekuceg broja
```

```
      tekuciBroj = 10 * tekuciBroj + s[i] - '0';
    else {
      // dosli smo do nekog operatora ili kraja broja

      // tekuci broj je faktor tekuceg sabirka
      tekuciSabirak *= tekuciBroj;
      // zavrsili smo sa obradom tekuceg broja i priremamo se za
      // citanje narednog
      tekuciBroj = 0;

      // ako smo stigli do kraja ili procitali aditivni operator
      // zavrsili smo obradu tekuceg sabirka
      if (i == s.length() || s[i] == '+' || s[i] == '-') {
        // rezultat uvecavamo ili umanjujemo za tekuci sabirak u
        // zavisnosti od ranije odredjenog znaka
        rezultat += znakTekucegSabirka * tekuciSabirak;
        if (i < s.length()) {
          // priremamo se za obradu narednog sabirka
          tekuciSabirak = 1;
          // njegov znak postavljamo u zavisnosti od operatora na koji
          // smo naisli
          znakTekucegSabirka = s[i] == '+' ? 1 : -1;
        }
      }
    }
  return rezultat;
}

int main() {
  string s;
  getline(cin, s);
  cout << vrednost(s) << endl;
  return 0;
}
```

Спекулативно израчунавање

Интересантна техника коју можемо применити у овом задатку је израчунавање редом једног по једног подизраза.

Размотримо израз `3+4-2+5*7`. Израчунавамо вредност израза `3`, затим `3+4`, затим `3+4-2`, затим `3+4-2+5` и на крају `3+4-2+5*7`. Вредност претходно израчунатог израза помаже да се израчуна вредност наредног израза. На пример, ако знамо вредност израза `3`, тада вредност израза `3+4` добијамо тако што на ту вредност додајемо вредност броја 4.

У општем случају, ако знамо вредност израза `e`, тада вредност израза `e+x` добијамо тако што на ту вредност додамо вредност броја `x`. Слично, ако знамо вредност израза `e`, тада вредност израза `e-x` добијамо тако што од те вредности одузмемо вредност броја `x`.

Случај множења је мало компликованији.

Ако знамо вредност израза `3+4-2+5`, вредност израза `3+4-2+5*7` не можемо израчунати једноставним множењем са 7. Наиме, у изразу `3+4-2+5` број 5 представља одређени вишак и он је додат на текућу суму спекулативно (под претпоставком да се иза њега неће наћи оператор множења). Ако се тај оператор ипак појави, онда ту вредност 5 треба одузети од текућег збира (тако да се добије вредност израза `3+4-2`), затим је помножити са 7 и на крају тај производ додати на вредност израза.

У општем случају, ако имамо израз облика `e+e'*x`, и знамо вредност израза `e+e'`, вредност новог израза добијамо тако што од вредности израза `e+e'` одузмемо вредност `e'` а затим додамо вредност `e'*x`. Да би ово било могуће уз вредност сваког текућег израза који мало по мало проширујемо, увек памтимо и вредност последњег сабирка који у њему учествује (тај сабирак може бити и негативан, ако је испред њега знак минус).

Прикажимо рад овог алгоритма на изразу `8+4*5*2-7*2`.

У старту вредност израза постављамо на нулу, као и вредност последњег сабирка. Након тога наилазимо на вредност `8`, увећавамо вредност израза на `8`, што је уједно и вредност последњег сабирка.

Након тога наилазимо на сабирање са вредношћу `4`, увећавамо вредност израза на `12`, а вредност последњег сабирка постављамо на `4`.

Пошто након тога наилазимо на множење бројем `5`, од вредности `12` одузимамо вредност `4` (која је погрешно додана) и додајемо `4*5`, чиме добијамо `28`, док вредност последњег сабирка постављамо на `20`.

Пошто наилазимо на још једно множење опет од вредности `28` одузимамо вредност `20` (која је погрешно додана), а затим додајемо `20*2` чиме добијамо вредност `48`, док вредност последњег сабирка постављамо на `40`.

Након тога долазимо до одузимања вредности `7` тако да вредност израза постаје `41`, а последњи сабирак постављамо на `-7`.

На крају, од збира одузимамо тих `-7` (који су погрешно додан) и увећавамо га за `-7*2` чиме добијамо `34`, а последњи сабирак постављамо на `-14`.

Коначна вредност израза је `34`.

```
#include <iostream>
#include <string>

using namespace std;

int vrednost(const string& s) {
  int rezultat = 0;
  char op = '+';
  int tekuciBroj = 0;
  int poslednjiSabirak = 0;

  for (int i = 0; i <= s.length(); i++)
    if (i < s.length() && isdigit(s[i]))
      // procitali smo cifru
      // dodajemo je kao poslednju cifru tekuceg broja
      tekuciBroj = 10 * tekuciBroj + s[i] - '0';
    else {
      switch (op) {
      case '+':
        // rezultat uvecavamo za tekuci broj (nadajuci se da iza njega
        // ne ide *)
        rezultat += tekuciBroj;
        poslednjiSabirak = tekuciBroj;
        break;
      case '-':
        // rezultat umanjujemo za tekuci broj (nadajuci se da iza
        // njega ne ide *)
        rezultat -= tekuciBroj;
        poslednjiSabirak = -tekuciBroj;
        break;
      case '*':
        // rezultat je u prethodnom koraku greskom uvecan za poslednji sabirak
        rezultat -= poslednjiSabirak;
        // poslednji sabirak treba da ukljuci i tekuci broj kao faktor
        poslednjiSabirak = poslednjiSabirak * tekuciBroj;
        // uvecavamo rezultat za azuirarani poslednji sabirak,
        // nadajuci se da se iza njega nece vise javljati znak *
        rezultat += poslednjiSabirak;
        break;
```

```
      }
      // zavrsili smo sa obradom tekuceg broja i pripremamo se za
      // citanje narednog
      tekuciBroj = 0;
      if (i < s.length())
        // pamtimo operator pre citanja narednog broja, jer nam on
        // govori kako naredni broj koji budemo procitali treba
        // ukljuciti u rezultat
        op = s[i];
    }
  return rezultat;
}

int main() {
  string s;
  getline(cin, s);
  cout << vrednost(s) << endl;
  return 0;
}
```

Задатак: Вредност израза

Написати програм којим се израчунавају и приказују вредности датих аритметичких израза. Сваки израз је исправно задат, састоји се од природних бројева и операција +, -, * и / (целобројно дељење). Коришћењем проширене Бекусове нотације (EBNF), синтаксу израза можемо описати на следећи начин:

```
<izraz>  ::= <term> {<operacija1> <term>}
<term>   ::= <faktor> {<operacija2> <faktor>}
<faktor> ::= <broj> | '(' <izraz> ')'
<broj>   ::= <cifra> {<cifra>}
<cifra> ::= '0' | '1' | ... | '9'
<operacija1> ::= '+' | '-'
<operacija2> ::= '*' | '/'
```

Улаз: У свакој линији стандарног улаза налази се исправан израз (израз не садржи размаке).

Излаз: Свака линија стандардног излаза садржи редом вредности израза датих на стандардном улазу, свака вредност у посебној линији. Ако израз није дефинисан, због дељења 0, приказати поруку `deljenje nulom`.

Пример

Улаз	*Излаз*
`1+2*3-4`	`3`
`2*3-5*(100-8*12)`	`-14`
`123-43*(12-3*5)/(17-35/2)`	`deljenje nulom`
`12/5+2`	`4`

Решење

Вредност израза рачунамо техником рекурзивног спуста. Сваки нетерминал граматике ћемо представити посебном функцијом која чита део израза који се извози из тог нетерминала и враћа вредност тог дела израза. Функцији се по референци преноси индекс `i` који означава почетак дела ниске `s` који се анализира. На крају рада функције овај индекс се премешта иза прочитаног дела ниске. Променљива `ok` која се такође преноси по референци је индикатор да ли је дошло до грешке дељења нулом током израчунавања вредности израза.

```
#include <iostream>
#include<string>

using namespace std;

int term(string, size_t&, bool&);
int faktor(string, size_t&, bool&);
int broj(string, size_t&);
```

```
int izraz(string s, size_t &i, bool &ok) {
  int a = term(s, i, ok);
  while (ok && i < s.length() && (s[i] == '+' || s[i] == '-')) {
    if (s[i] == '+') {
      i++;
      a += term(s, i, ok);
    } else if (s[i] == '-') {
      i++;
      a -= term(s,i,ok);
    }
  }
  return a;
}

int term(string s, size_t &i, bool &ok) {
  int a = faktor(s,i,ok);
  while (ok && i < s.length() && (s[i] == '*' || s[i] == '/')) {
    i++;
    if (s[i - 1] == '*')
      a *= faktor(s,i,ok);
    else {
      int b = faktor(s,i,ok);
      if (b == 0)
        ok = false;
      else
        a /= b;
    }
  }
  return a;
}

int faktor(string s, size_t&i, bool &ok) {
  if (isdigit(s[i]))
    return broj(s, i);
  else {
    i++;
    int a = izraz(s, i, ok);
    i++;
    return a;
  }
}

int broj(string s, size_t &i) {
  int x = 0;
  while (i < s.length() && isdigit(s[i])) {
    x = x*10 + s[i]-'0';
    i++;
  }
  return x;
}

int main() {
  string s;
  size_t i;
  int rez;
  bool ok;
  while (cin >> s) {
```

```
    i = 0;
    ok = true;
    rez = izraz(s, i, ok);
    if (ok)
      cout << rez << endl;
    else
      cout << "deljenje nulom" << endl;
  }
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Вредност потпуно заграђеног мин-макс израза

Мин-макс изрази се граде применом две инфиксно записане операције:

- операција `m` означава проналажење минимума два броја;
- операција `M` означава проналажење максимума два броја.

Напиши програм који израчунава вредност датог израза у коме је свака примена операције ограђена заградама (дакле израз је или једна цифра или је облика `(<izraz> op <izraz>)`).

Улаз: Једина линија стандардног улаза садржи исправан потпуно заграђени мин-макс израз.

Излаз: На стандардни излаз исписати цифру која представља вредност учитаног израза.

Пример

Улаз	*Излаз*
`((3m5)M(4m(2M6)))`	`4`

Објашњење

`((3m5)M(4m(2M6))) = (3M(4m6)) = (3M4) = 4`

Види другачија решења овог задатка.

Задатак: Вредност мин-макс израза

Изрази се граде од цифара, заграда и два инфиксна оператора. Оператор `m` означава проналажење минимума два броја, а `M` максимума два броја. Оператор `M` има већи приоритет. Напиши програм који израчунава вредност учитаног израза.

Улаз: Са стандардног улаза се учитава израз описаног облика (у једној линији, без размака).

Излаз: На стандардни излаз исписати вредност учитаног израза.

Пример 1

Улаз	*Излаз*
`3m4M5`	`3`

Пример 2

Улаз	*Излаз*
`((((9M1)M(0m6)m0M0)m0M6m(9m1M3m(4m(6m8))))M8M5)m(1M2M(0m6))`	`2`

Пример 3

Улаз	*Излаз*
`7`	`7`

Глава 4

Генерисање комбинаторних објеката

У овој глави биће приказано како је могуће набројати све објекте који имају неку задату комбинаторну структуру. У већини задатака могуће је разматрати две врсте решења. Једна група решења је заснована на рекурзивном поступку набрајања објеката, док је друга група решења заснована на проналажењу наредног комбинаторног објекта у односу на неки задати редослед (најчешће лексикорафски).

Задатак: Следећа варијација

Напиши програм који одређује наредну варијацију дужине k скупа $\{1, \ldots, n\}$ у лексикографском поретку.

Улаз: Прва линија стандардног улаза садржи број k ($1 \leq k \leq 100$), а друга број n ($1 \leq n \leq 100$). У трећој линији се налази варијација описана бројевима раздвојеним по једним размаком.

Излаз: На стандардни излаз исписати следећу варијацију у лексикографском поретку, ако она постоји, или `-`, ако је учитана варијација лексикографски максимална.

Пример

Улаз	*Излаз*
5 4 1 1 2 3 4	1 1 2 4 1

Решење

Следећа варијација у лексикографском поретку се може генерисати тако што се увећа последњи број у варијацији који се може увећати, и што се након увећавања сви бројеви иза увећаног броја поставе на 1. Позиција на којој се број увећава назива се *преломна тачка* (енгл. turning point). На пример, ако набрајамо варијације скупа $\{1, 2, 3\}$ дужине 5 наредна варијација за варијацију 21332 је 21333 (преломна тачка је позиција 4, која је последња позиција у низу), док је њој наредна варијација 22111 (преломна тачка је позиција 1 на којој се налазио елемент 1). Низ 33333 нема преломну тачку, па самим тим ни лексикографски следећу варијацију.

Један начин имплементације је да преломну тачку нађемо линеарном претрагом од краја низа, ако преломна тачка постоји да увећамо елемент и да од следеће позиције до краја низ попунимо јединицама. Међутим, те две фазе можемо објединити. Варијацију обилазимо од краја постављајући на 1 сваки елемент у варијацији који је једнак броју n. Ако се зауставимо пре него што смо стигли до краја низа, значи да смо пронашли елемент који се може увећати и увећавамо га. У супротном је варијација имала све елементе једнаке n и била је максимална у лексикографском редоследу.

```
#include <iostream>
#include <vector>

using namespace std;

void ispisi(const vector<int>& varijacija) {
    for (int x : varijacija)
      cout << x << " ";
```

```
    cout << endl;
}

bool sledecaVarijacija(int n, vector<int>& varijacija) {
   // od kraja varijacije tražimo prvi element koji se moze povecati
   int i;
   int k = varijacija.size();
   for (i = k-1; i >= 0 && varijacija[i] == n; i--)
      varijacija[i] = 1;
   // svi elementi su jednaki n - ne postoji naredna varijacija
   if (i < 0) return false;
   // uvecavamo element koji je moguće uvecati
   varijacija[i]++;
   return true;
}

int main() {
  int k, n;
  cin >> k >> n;
  vector<int> varijacija(k);
  for (int i = 0; i < k; i++)
    cin >> varijacija[i];
  if (sledecaVarijacija(n, varijacija))
    ispisi(varijacija);
  else
    cout << "-" << endl;
  return 0;
}
```

Задатак: Све варијације

Напиши програм који одређује све варијације са понављањем дужине k скупа $\{1, \ldots, n\}$.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 5$) и у наредној линији број k ($1 \leq k \leq 5$).

Излаз: На стандардни излаз исписати све варијације, сортиране лексикографски.

Пример

Улаз	*Излаз*
2	1 1 1
3	1 1 2
	1 2 1
	1 2 2
	2 1 1
	2 1 2
	2 2 1
	2 2 2

Решење

Рекурзивно генерисање варијација

Варијације се могу набројати индуктивно рекурзивном конструкцијом. Једина варијација дужине нула је празна. Све варијације дужине k се могу добити тако што се на прво место упише било који од бројева од 1 до n, а затим се преостала места допуне свим варијацијама дужине $k-1$. Имплементацију ћемо организовати тако да уместо да враћа колекцију варијација, рекурзивна функција прима делимично попуњен низ који ће на све могуће начине допуњавати варијацијама текуће дужине k (која ће се смањивати кроз рекурзивне позиве). Дакле, на текућу позицију у низу (она се израчунава као разлика између дужине низа и тренутне вредности k) постављамо једну по једну вредност од 1 до n и затим рекурзивно позивамо функцију да попуни остатак низа (тиме што смањујемо дужину k и тиме прелазимо на наредну позицију).

Рецимо и да је могуће на последње место постављати један по један број од 1 до n, а затим рекурзивно попуњавати префикс, но тиме би редослед варијација био другачији од траженог.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje tekucu varijaciju na standardni izlaz
void obradi(const vector<int>& varijacija) {
  for (int x : varijacija)
    cout << x << " ";
  cout << endl;
}

// sve varijacije duzine k elemenata skupa {1, ..., n}
// Dati niz varijacija duzine varijacije.size() - k
// se dopunjuje svim mogucim varijacijama sa ponavljanjem
// duzine k skupa {1, ..., n} i sve tako
// dobijene varijacije se obrađuju
void obradiSveVarijacije(int k, int n,
                         vector<int>& varijacija) {
  // k je 0, pa je jedina varijacija duzine nula prazna i njenim
  // dodavanjem na polazni niz on se ne menja
  if (k == 0)
    obradi(varijacija);
  else
    // na tekucu poziciju postavljamo sve moguce vrednosti od 1 do n i
    // dobijeni niz onda rekurzivno proširujemo
    for (int nn = 1; nn <= n; nn++) {
       varijacija[varijacija.size() - k] = nn;
       obradiSveVarijacije(k-1, n, varijacija);
    }
}


// sve varijacije duzine k skupa {1, ..., n}
void obradiSveVarijacije(int k, int n) {
  vector<int> varijacija(k);
  obradiSveVarijacije(k, n, varijacija);
}


int main() {
  int n, k;
  cin >> n >> k;
  obradiSveVarijacije(k, n);
  return 0;
}
```

Проналажење лексикографски следеће варијације

Друга могућност је да се крене од лексикографски најмање варијације (то је варијација $\underbrace{11\ldots11}_{k}$) и да се коришћењем функције описане у задатку Следећа варијација одређује наредна варијација дате варијације у односу на лексикографски редослед, све док таква постоји.

```
#include <iostream>
#include <vector>
```

```
using namespace std;

// ispisuje tekucu varijaciju na standardni izlaz
void obradi(const vector<int>& varijacija) {
  for (int x : varijacija)
    cout << x << " ";
  cout << endl;
}

bool sledecaVarijacija(int n,
                       vector<int>& varijacija) {
   // od kraja varijacije tražimo prvi element koji se moze povecati
   int i;
   int k = varijacija.size();
   for (i = k-1; i >= 0 && varijacija[i] == n; i--)
      varijacija[i] = 1;
   // svi elementi su jednaki n - ne postoji naredna varijacija
   if (i < 0) return false;
   // uvecavamo element koji je moguće uvecati
   varijacija[i]++;
   return true;
}

void obradiSveVarijacije(int k, int n) {
   // krećemo od varijacije 11...11 - ona je leksikografski najmanja
   vector<int> varijacija(k, 1);
   // obradjujemo redom varijacije dok god postoji leksikografski
   // sledeca
   do {
     obradi(varijacija);
   } while(sledecaVarijacija(n, varijacija));
}

int main() {
  int n, k;
  cin >> n >> k;
  obradiSveVarijacije(k, n);
  return 0;
}
```

Задатак: Све речи од датих слова

Стрингом s дат је скуп малих слова енглеског алфабета (слова су у стрингу уређена у растућем поретку) и природан број k. Написати програм којим се приказују у лексикографском поретку све речи дужине k које се могу формирати од датог скупа.

Улаз: На стандардном улазу у првој линији налази се стринг s дужине највише 10, у другој линији налази се природан број k ($k \leq 6$, $k \leq n$).

Излаз: На стандардном излазу приказати тражене речи у лексикографском поретку, сваку реч у посебној линији.

Пример

Улаз	*Излаз*
amx	aa
2	am
	ax
	ma
	mm
	mx
	xa
	xm
	xx

Решење

Задатак можемо решити веома слично задатку Све варијације. Дефинишемо рекурзивну функцију која прима скуп слова, реч која се мало по мало попуњава и индекс наредне позиције i у тој речи коју треба попунити (иницијално је та позиција једнака нули). Када је позиција једнака дужини речи, реч је цела формирана и може се исписати. У супротном на позицију i постављамо редом једно по једно слово из датог скупа и рекурзивно позивамо функцију да се попуне слова од позиције $i + 1$, надаље.

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

void sveReci(string& s, const string& slova, int i) {
  if (i == s.length())
    cout << s << endl;
  else
    for (char slovo : slova) {
      s[i] = slovo;
      sveReci(s, slova, i+1);
    }
}

void sveReci(const string& slova, int k) {
  string s;
  s.resize(k);
  sveReci(s, slova, 0);
}

int main() {
  ios_base::sync_with_stdio(false);
  string slova;
  int k;
  cin >> slova;
  cin >> k;
  sveReci(slova, k);
  return 0;
}
```

Задатак: Сви подскупови

Напиши програм који исписује све подскупове датог скупа.

Улаз: Са стандардног улаза се учитава број n (важи $3 \leq n \leq 10$), а затим n природних бројева, растуће сортираних, раздвојених по једним размаком.

Излаз: На стандардни излаз исписати све подскупове учитаног скупа бројева, сваки у посебном реду, са елементима раздвојеним једним размаком. Прво се ређају подскупови у којима први елемент није укључен,

а затим они у којима јесте. У свакој од те две групе, прво се исписују подскупови у којима други елемент није укључен, а затим они где јесте и тако даље.

Пример

Улаз	*Излаз*
3	3
1 2 3	2
	2 3
	1
	1 3
	1 2
	1 2 3

Решење

Генерисање свих подскупова одговара генерисању свих варијација дужине n од нула и јединица (сваки елемент је или укључен или искључен), па је решење слично оном приказаном у задатку Све варијације.

Иако многи савремени језици пружају тип за репрезентовање скупова, имплементација је једноставнија и ефикаснија ако се елементи скупа чувају у низу. Да бисмо избегли потребу за продужавањем и скраћивањем низа, низ можемо алоцирати на максималну могућу дужину (број елемената полазног скупа) и паралелно са низом можемо одржавати број елемената подскупа који је тренутно смештен у низ (он је скоро увек строго мањи од дужине низа).

Ако је скуп S празан, онда је једини његов подскуп празан, а ако није, онда се може разложити на неки елемент x и скуп $S' = S \setminus x$ добијен када се тај елемент избаци из полазног скупа. Пошто је скуп S' мањи од скупа S, његови се подскупови могу одредити рекурзивно. Сви подскупови полазног скупа S су онда они који су одређени за мањи скуп S', као и сви они који се од њих добијају додавањем издвојеног елемента x. Ову конструкцију није економично програмски реализовати, јер се претпоставља да резултат рада функције представља скуп свих подскупова скупа. Уместо такве функције дефинисаћемо процедуру која неће истовремено чувати и враћати све подскупове већ само један по један набројати и обрадити. До решења се може доћи тако што се у рекурзивном позиву проследи парцијално попуњени подскуп који или не садржи или садржи издвојени елемент, а рекурзивни позив има задатак да онда подскуп који је примио на све могуће начине допуни подскуповима смањеног скупа S'.

Дефинишемо рекурзивну функцију која на сваком наредном нивоу рекурзије обрађује наредни елемент полазног скупа (представљеног низом). У првом случају га не додаје у резултујући подскуп (такође представљен низом, који прослеђујемо као додатни параметар) и прелази на наредни ниво рекурзије, а у другом га додаје на крај тренутног резултујућег подскупа и прелази на наредни ниво рекурзије. Када се цео полазни низ исцрпи (када је дубина рекурзије једнака дужини полазног низа), тада се тренутно акумулирани подскуп исписује.

```
#include <iostream>
#include <vector>

using namespace std;

// ispis prvih n elemenata niza a
void ispisi(const vector<int>& a, int n) {
  for (int k = 0; k < n; k++)
    cout << a[k] << " ";
  cout << endl;
}

// procedura određuje i obrađuje sve moguće skupove koji se dobijaju
// tako što se na elemente prosleđenog podskupa p dužine j, dodaju
// podskupovi prosleđenog skupa smeštenog u nizu a, od pozicije i nadalje
void ispisi_sve_podskupove(const vector<int>& a, int i, vector<int>& p, int j) {
  // skup preostalih elemenata u nizu a koji se mogu ubaciti u podskup je prazan
  if (i == a.size())
    // ispisujemo formirani podskup
    ispisi(p, j);
```

```
  else {
    // element na poziciji i ne uključujemo u podskup
    ispisi_sve_podskupove(a, i + 1, p, j);
    // element na poziciji i uključujemo u podskup
    p[j] = a[i];
    ispisi_sve_podskupove(a, i + 1, p, j + 1);
  }
}

void ispisi_sve_podskupove(const vector<int>& a) {
  // podskup je na početku prazan, i u njega potencijalno dodajemo sve
  // elemente skupa a od pozicije 0 nadalje
  vector<int> p(a.size());
  ispisi_sve_podskupove(a, 0, p, 0);
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  ispisi_sve_podskupove(a);
  return 0;
}
```

Једно решење је да се у посебном низу логичких вредности набрајају све варијације скупа тачно-нетачно. Свака таква варијација одговара једном подскупу, тако што се у подскуп укључују елементи са оних позиција на којима је је вредност тачно. Варијације набрајамо коришћењем функције за одређивање следеће варијације, описане у задатку Следећа варијација.

```
#include <iostream>
#include <vector>

using namespace std;

void ispisi(const vector<bool>& v, const vector<int>& a) {
  for (int i = 0; i < v.size(); i++)
    if (v[i])
      cout << a[i] << " ";
  cout << endl;
}

bool sledecaVarijacija(vector<bool>& v) {
  int i = v.size() - 1;
  while (i >= 0 && v[i])
    v[i--] = false;
  if (i < 0) return false;
  v[i] = true;
  return true;
}

void ispisiSvePodskupove(const vector<int>& a) {
  vector<bool> v(a.size(), false);
  do {
    ispisi(v, a);
  } while (sledecaVarijacija(v));
}
```

```
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  ispisiSvePodskupove(a);
  return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// ispis elemenata niza a
void ispisi(const vector<int>& a) {
  for (int i = 0; i < a.size(); i++)
      cout << a[i] << " ";
  cout << endl;
}

// procedura određuje i obrađuje sve moguće skupove koji se dobijaju
// tako što se na elemente prosleđenog podskupa dodaju podskupovi
// prosleđenog skupa
void obradiSvePodskupove(const vector<int>& skup, const vector<int>& podskup) {
  // skup je prazan
  if (skup.size() == 0)
    // na prosleđeni podskup možemo dodati samo prazan skup
    ispisi(podskup);
  else {
    // izdvajamo i uklanjamo proizvoljan element skupa
    int x = skup.back();
    vector<int> smanjenSkup = skup;
    smanjenSkup.pop_back();
    // u podskup dodajemo sve podskupove skupa bez izdvojenog
    // elementa
    vector<int> podskupBez = podskup;
    obradiSvePodskupove(smanjenSkup, podskupBez);
    // u podskup uključujemo izdvojeni element i zatim sve
    // podskupove skupa bez izdvojenog elementa
    vector<int> podskupSa = podskup;
    podskupSa.push_back(x);
    obradiSvePodskupove(smanjenSkup, podskupSa);
  }
}

void obradiSvePodskupove(const vector<int>& skup) {
  // pošto elementi iz skupa u podskup prebacuju sa desnog kraja, da
  // bismo dobili traženi redosled podskupa, potrebno je da obrnemo
  // redosled elemenata skupa
  vector<int> skupObratno = skup;
  reverse(begin(skupObratno), end(skupObratno));
  // krećemo od praznog podskupa
  vector<int> podskup;
  // prazan skup proširujemo svim podskupovima datog skupa
  obradiSvePodskupove(skupObratno, podskup);
}
```

```
int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  obradiSvePodskupove(a);
  return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// ispis vektora
void ispisi(const vector<int>& a) {
  for (int i = 0; i < a.size(); i++)
      cout << a[i] << " ";
  cout << endl;
}

// procedura određuje i obrađuje sve moguće skupove koji se dobijaju
// tako što se na elemente prosleđenog podskupa dodaju podskupovi
// prosleđenog skupa
void obradiSvePodskupove(vector<int>& skup, vector<int>& podskup) {
  // skup je prazan
  if (skup.size() == 0)
    // na prosleđeni podskup možemo dodati samo prazan skup
    ispisi(podskup);
  else {
    // izdvajamo i uklanjamo proizvoljan element skupa
    int x = skup.back();
    skup.pop_back();
    // u podskup dodajemo sve podskupove skupa bez izdvojenog
    // elementa
    obradiSvePodskupove(skup, podskup);
    // u podskup uključujemo izdvojeni element i zatim sve
    // podskupove skupa bez izdvojenog elementa
    podskup.push_back(x);
    obradiSvePodskupove(skup, podskup);
    // vraćamo skup i podskup u početno stanje
    podskup.pop_back();
    skup.push_back(x);
  }
}

void obradiSvePodskupove(vector<int>& skup) {
  // pošto elementi iz skupa u podskup prebacuju sa desnog kraja, da
  // bismo dobili traženi redosled podskupa, potrebno je da obrnemo
  // redosled elemenata skupa
  vector<int> skupObratno = skup;
  reverse(begin(skupObratno), end(skupObratno));
  // krećemo od praznog podskupa
  vector<int> podskup;
  // efikasnosti radi rezervišemo potrebnu memoriju za najveći podskup
  podskup.reserve(skup.size());
```

```
  // prazan skup proširujemo svim podskupovima datog skupa
  obradiSvePodskupove(skupObratno, podskup);
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  obradiSvePodskupove(a);
  return 0;
}
```

Задатак: Следећи подскуп

Напиши програм који одређује подскуп скупа бројева $\{1, \ldots, n\}$ који у лексикографском редоследу следи непосредно иза датог подскупа. Подскупови су задати у облику строго растуће сортираних низова.

Улаз: Прва линија садржи број n ($1 \leq n \leq 100$), а наредна линија садржи подскуп чији су елементи задати сортирано растуће, раздвојени по једним размаком.

Излаз: На стандардни излаз у једној линији исписати елементе траженог подскупа тј. - ако је учитани подскуп лексикографски највећи.

Пример

Улаз	*Излаз*
5 1 2 3 4 5	1 2 3 5

Решење

Напишимо, на пример, лексикографски уређен списак свих подскупова скупа бројева од 1 до 4.

```
-
1
12
123
1234
124
13
134
14
2
23
234
24
3
34
4
```

Можемо приметити да постоје два начина да се дође до наредног подскупа. Анализирајмо ове скупове у истом редоследу, груписане и на основу броја елемената.

```
_ 1  12  123  1234
         124
     13  134
     14
  2  23  234
     24
  3  34
  4
```

Један начин је *проширивање* када се наредни подскуп добија додавањем неког елемента у претходни. То су кораци у претходној табели код којих се прелази из једне у наредну колону. Да би добијени подскуп следио непосредно иза претходног у лексикографском редоследу, додати елемент подскупу мора бити најмањи могући. Пошто је сваки подскуп сортиран, елемент мора бити за један већи од последњег елемента подскупа који се проширује (изузетак је празан скуп, који се проширује елементом 1). Једини случај када проширивање није могуће је када је последњи елемент подскупа највећи могући (у нашем примеру то је 4).

Други начин је *скраћивање* када се наредни елемент добија уклањањем неких елемената из подскупа и изменом преосталих елемената. То су кораци у претходној табели код којих се прелази са краја једне у наредну врсту. У овом случају скраћивање функционише тако што се из подскупа избаци завршни највећи елемент, а затим се највећи од преосталих елемената увећа за 1 (он не може бити највећи, јер су елементи унутар сваког подскупа строго растући). Ако након избацивања највећег елемента остане празан скуп, наредна комбинација не постоји.

Подскупове можемо представити динамичким низом који нам омогућава да елементе додајемо и уклањамо са десног краја. У језику C++ можемо употребити вектор (колекцију `vector` из заглавља `<vector>`).

```
#include <iostream>
#include <vector>

using namespace std;

// na osnovu datog podskupa skupa {1, ..., n} određuje
// leksikografski naredni podskup i vraća da li takav
// podskup postoji
bool sledeciPodskup(vector<int>& podskup, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (podskup.empty()) {
    podskup.push_back(1);
    // podskup je uspešno pronađen
    return true;
  }
  // proširivanje
  if (podskup.back() < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup.push_back(podskup.back() + 1);
    // podskup je uspešno pronađen
    return true;
  }

  // skraćivanje
  // uklanjamo poslednji najveći element
  podskup.pop_back();
  // ako nema preostalih elemenata ne postoji naredni podskup
  if (podskup.empty())
    return false;
  // najveći od preostalih elemenata uvećavamo za 1
  podskup.back()++;
  // podskup je uspešno pronađen
  return true;
}

int main() {
  int n;
  cin >> n;
  vector<int> podskup;
  int x;
  while (cin >> x)
    podskup.push_back(x);
```

```
  if (sledeciPodskup(podskup, n)) {
    for (int x : podskup)
      cout << x << " ";
    cout << endl;
  } else {
    cout << "-" << endl;
  }
  return 0;
}
```

Подскупове можемо чувати и у оквиру низа који је унапред алоциран тако да може да смести елементе највећег подскупа (оног који има тачно n елемената). У том случају је неопходно да одржавамо и промењиву у којој бележимо број елемената подскупа. Пошто се она мења у функцији која одређује наредни подскуп, потребно је пренети је по референци.

```
#include <iostream>

using namespace std;

// na osnovu datog podskupa skupa {1, ..., n} određuje
// leksikografski naredni podskup i vraća da li takav
// podskup postoji.
// Tekući podskup je smešten u nizu dužine k
bool sledeciPodskup(int podskup[], int& k, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (k == 0) {
    podskup[k++] = 1;
    return true;
  }

  // proširivanje
  if (podskup[k-1] < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup[k] = podskup[k-1] + 1;
    k++;
    return true;
  }

  // skraćivanje

  // izbacujemo najveći element iz podskupa
  k--;
  // ako nema preostalih elemenata, naredni podskup ne postoji
  if (k == 0)
    return false;

  // najveći od preostalih elemenata uvećavamo za 1
  podskup[k-1]++;
  return true;
}


int main() {
  int n;
  cin >> n;
  // elementi podskupa
  const int MAX = 100;
  int podskup[MAX];
```

```
  // broj elemenata podskupa
  int k = 0;
  // ucitavamo elemente datog podskupa
  int x;
  while (cin >> x)
    podskup[k++] = x;
  // pokusavamo da pronadjemo naredni podskup
  if (sledeciPodskup(podskup, k, n)) {
    // ako postoji, ispisujemo ga
    for (int i = 0; i < k; i++)
      cout << podskup[i] << " ";
    cout << endl;
  } else
    // naredni podskup ne postoji
    cout << "-" << endl;
  return 0;
}
```

Задатак: Сви подскупови лексикографски

Напиши програм који исписује све подскупове скупа $\{1, \ldots, n\}$ у лексикографском редоследу.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 15$).

Излаз: На стандардни излаз исписати тражене подскупове, сваки у посебном реду. Сваки се подскуп представља растуће сортираним низом својих елемената.

Пример

Улаз	*Излаз*
3	1
	1 2
	1 2 3
	1 3
	2
	2 3
	3

Решење

Задатак се једноставно може решити коришћењем функције за одређивање следећег подскупа у лексикографском редоследу, описане у задатку Следећи подскуп.

```
#include <iostream>
#include <vector>

using namespace std;

void ispisi(const vector<int>& v, int k) {
  for (int i = 0; i < k; i++)
    cout << v[i] << " ";
  cout << endl;
}

// na osnovu datog podskupa skupa {1, ..., n} određuje
// leksikografski naredni podskup i vraća da li takav
// podskup postoji.
// Tekući podskup je smešten u nizu dužine k
bool sledeciPodskup(vector<int>& podskup, int& k, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (k == 0) {
    podskup[k++] = 1;
```

```
    return true;
  }

  // proširivanje
  if (podskup[k-1] < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup[k] = podskup[k-1] + 1;
    k++;
    return true;
  }

  // skraćivanje

  // izbacujemo najveći element iz podskupa
  k--;
  // ako nema preostalih elemenata, naredni podskup ne postoji
  if (k == 0)
    return false;

  // najveći od preostalih elemenata uvećavamo za 1
  podskup[k-1]++;
  return true;
}

void ispisiSvePodskupove(int n) {
  vector<int> podskup(n);
  int k = 0;
  do {
    ispisi(podskup, k);
  } while (sledeciPodskup(podskup, k, n));
}


int main() {
  ios_base::sync_with_stdio(false); cout.tie(0);
  int n;
  cin >> n;
  ispisiSvePodskupove(n);
  return 0;
}
```

Задатак: Следећи бинарни низ без суседних јединица

Посматрајмо лексикографски поређане бинарне низове дужине n који садрже нуле и јединице, али не садрже две суседне јединице. На пример, такви низови дужине 3 су 000, 001, 010, 100 и 101. Напиши програм који за дати низ одређује наредни низ у лексикографском поретку.

Улаз: Са стандардног улаза се учитава бинарни низ без узастопних јединица дужине n ($1 \leq n \leq 50$). Сви елементи су записани један иза другог, без размака.

Излаз: Једина линија стандардног излаза треба да садржи елементе наредног низа у лексикографском поретку (исписане један иза другог, без размака) или текст `ne postoji` ако је учитани низ лексикографски највећи.

Пример 1

Улаз	*Излаз*
`10101000100001010`	`10101000100010000`

Пример 2

Улаз	*Излаз*
`10101010`	`ne postoji`

Решење

Алгоритам којим се одређује следећи низ у лексикографском редоследу представљаће модификацију алгоритма којим се одређује следећа варијација у лексикографском редоследу, описан у задатку Следећа варијација. Подсетимо се, крећемо од краја низа, проналазимо прву позицију на којој се налази елемент чија вредност није тренутно максимална (ако она постоји), увећавамо је, и након тога све елементе иза ње постављамо на минималну вредност (у нашем контексту максимална вредност је 1, а минимална 0). Ово можемо остварити и у једном проласку тако што крећући се уназад све максималне вредности одмах постављамо на нулу.

Модификујмо сада овај алгоритам тако да ради за низове који немају две узастопне јединице. Ако се након примене претходног алгоритма догодило да је увећана (постављена на јединицу) цифра испред које не стоји јединица, то је решење које смо тражили. На пример, следећи низ у односу на низ 01001 је 01010. Међутим, ако затражимо наредни елемент, добићемо 01011, а то је елемент који није допуштен. Настављањем даље добијамо 01100, што је такође елемент који није допуштен, након тога ређају се елементи од 01101, до 01111, који су сви недопуштени да бисмо на крају добили 10000, што је заправо наредни елемент у односу на 01010. Дакле, опет се крећемо са краја низа и уписујемо нуле све док се на тренутној или на претходној позицији у низу налази јединица. На крају, на позицији на којој смо се зауставили и нисмо уписали нулу (ако таква постоји) уписујемо јединицу (то је позиција на којој пише нула и испред ње или нема ништа или пише нула). Ако таква позиција не постоји, онда је тренутни низ лексикографски највећи.

Једноставности ради, низ представљамо у облику ниске карактера. Наравно, могућа је и репрезентација у неком облику низа целих бројева.

```
#include <iostream>
#include <string>

using namespace std;

bool sledeciNiz(string& s){
  int n = s.length();
  int i = n - 1;
  while ((i >= 0  && s[i] == '1') ||
         (i > 0 && s[i - 1] == '1'))
    s[i--] = '0';
  if (i < 0)
    return false;
  s[i] = '1';
  return true;
}

int main() {
  string s;
  cin >> s;
  if (sledeciNiz(s))
    cout << s << endl;
  else
    cout << "ne postoji" << endl;
  return 0;
}
```

Задатак: Сви бинарни низови без суседних јединица

Напиши програм који исписује све низове бинарних бројева дате дужине у којима се не јављају две узастопне јединице. Бројеве исписати у лексикографском редоследу.

Улаз: Са стандардног улаза се уноси број n.

Излаз: На стандардни излаз исписати тражене бројеве, сваки у посебном реду.

Пример

Улаз	*Излаз*
3	000
	001
	010
	100
	101

Решење

Задатак представља модификацију задатка Све варијације.

Један начин је да дефинишемо рекурзивну функцију која генерише тражене бројеве. Она добија префикс дужине i и покушава на све начине да га прошири (кроз рекурзију проширујемо низ карактера у старту алоциран на дужину n дужину i његовог попуњеног дела). Ако је $i = n$, тада је цео низ попуњен и исписује се. У супротном, на позицију i увек можемо дописати нулу и рекурзивно наставити са продужавањем тако добијене ниске. Са друге стране, јединицу можемо уписати само ако претходни карактер није јединица (у супротном бисмо добили две узастопне јединице). То се дешава или када нема претходне цифре (када је $i = 0$) или када је претходна цифра (на позицији $i - 1$) различита од јединице.

```
#include <iostream>
#include <string>

using namespace std;

void obradi(const string& binarni) {
  cout << binarni << endl;
}

void obradiSveBinarneBez11(string& binarni, int i) {
  if (i == binarni.size())
    obradi(binarni);
  else {
    binarni[i] = '0';
    obradiSveBinarneBez11(binarni, i+1);
    if (i == 0 || binarni[i-1] != '1') {
      binarni[i] = '1';
      obradiSveBinarneBez11(binarni, i+1);
    }
  }
}

void obradiSveBinarneBez11(int n) {
  string binarni(n, '0');
  obradiSveBinarneBez11(binarni, 0);
}

int main() {
  int n;
  cin >> n;
  obradiSveBinarneBez11(n);
  return 0;
}
```

Једна могућност је да се употреби функција која генерише наредну у лексикографском поретку бинарну ниску без суседних јединица (та функција је описана у задатку Следећи бинарни низ без суседних јединица). Креће се од ниске која садржи n нула и исписује се и рачуна наредна варијација све док таква постоји.

```
#include <iostream>
#include <string>

using namespace std;
```

```
void obradi(const string& binarni) {
  cout << binarni << endl;
}

bool sledeciBinarniBez11(string& s){
  int n = s.length();
  int i = n - 1;
  while ((i >= 0  && s[i] == '1') ||
         (i > 0 && s[i - 1] == '1'))
    s[i--] = '0';
  if (i < 0)
    return false;
  s[i] = '1';
  return true;
}

void obradiSveBinarneBez11(int n) {
  string binarni(n, '0');
  do {
    obradi(binarni);
  } while (sledeciBinarniBez11(binarni));
}


int main() {
  int n;
  cin >> n;
  obradiSveBinarneBez11(n);
  return 0;
}
```

Задатак: Бројеви који у бинарном запису немају две суседне нуле

Написати програм којим се приказују декадни записи свих природних бројева који у бинарном систему имају највише n бинарних цифара и немају две узастопне нуле.

Улаз: Прва линија стандардног улаза садржи природан број n ($1 \leq n \leq 20$).

Излаз: На стандардном излазу приказати тражене бројеве у растућем поретку, сваки број у посебној линији.

Пример

Улаз	*Излаз*
3	1
	2
	3
	5
	6
	7

Решење

Овај задатак је веома сличан задатку Сви бинарни низови без суседних јединица и може се решавати аналогно њему. Основа разлика је то што је након генерисања бинарног низа потребно израчунати његову декадну вредност. То можемо урадити Хорнеровом шемом, како је приказано у задатку Број формиран од датих цифра с лева на десно.

```
#include <iostream>
#include <vector>

using namespace std;
```

```
int binUDekadni(const vector<bool>& b) {
  int d = 0;
  for (bool x : b)
    d = 2 * d + (x ? 1 : 0);
  return d;
}

void generisi_(vector<bool>& tekuci, int poz) {
  if (poz == tekuci.size()) {
    cout << binUDekadni(tekuci) << endl;
    return;
  }
  if (poz > 0 && tekuci[poz-1] != false) {
    tekuci[poz] = false;
    generisi_(tekuci, poz + 1);
  }
  tekuci[poz] = true;
  generisi_(tekuci, poz + 1);
}

void generisi(int n) {
  for (int brojBinCifara = 1; brojBinCifara <= n; brojBinCifara++) {
    vector<bool> tekuci(brojBinCifara);
    generisi_(tekuci, 0);
  }
}

int main() {
  int n;
  cin >> n;
  generisi(n);
  return 0;
}
```

Још једна могућност је да уместо у облику низа цифара (низа података типа `char`, `int` или `bool`) број памтимо у облику једног целобројног податка (најбоље типа `unsigned`), чији интерни бинарни запис представља наш текући низ цифара. Проверу последње цифре можемо извршити битовском конјункцијом (`&`) са бројем 1 (последња цифра је 1 ако и само ако се добије резултат 1, тј. вредност различита од 0), додавање нуле на десни крај можемо остварити померањем (шифтовањем) улево за једно место (оператором `<<`), а додавање јединице можемо остварити шифтовањем улево за једном место и затим израчунавањем битовске дисјункције (`|`) са бројем 1. Овим се избегава потреба за прерачунавањем низа цифара у декадни систем (исписивањем података типа `unsigned` исписује се аутоматски његова декадна вредност). Нагласимо и да је функцију генерисања потребно посебно позивати за сваки број цифара од 1 па до унетог максималног броја цифара n.

```
#include <iostream>

using namespace std;

void generisi_(unsigned tekuci, int preostaloCifara) {
  if (preostaloCifara == 0) {
    cout << tekuci << endl;
    return;
  }
  if ((tekuci & 1) != 0)
    generisi_(tekuci << 1, preostaloCifara - 1);
  generisi_((tekuci << 1) | 1, preostaloCifara - 1);
}
```

```
void generisi(int n) {
  for (int brojBinCifara = 1; brojBinCifara <= n; brojBinCifara++)
    generisi_(0, brojBinCifara);
}

int main() {
  int n;
  cin >> n;
  generisi(n);
  return 0;
}
```

Задатак: Следећа комбинација

Комбинације дужине k од n елемената подразумевају да се врши одабир k елемената скупа $\{1, \ldots, n\}$, слично као што се, на пример, у игри лото бира 7 од 39 куглица. Напиши програм који за дату комбинацију одређује наредну у лексикографском поретку.

Улаз: Са стандардног улаза се уноси број n ($2 \leq n \leq 100$) а затим у наредном реду једна комбинација дужине $1 \leq k \leq n$. Елементи су задати сортирани растуће, одвојени по једним размаком.

Излаз: На стандардни излаз исписати комбинацију која је наредна у лексикографском редоследу у односу на дату, тј. - ако таква комбинација не постоји.

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
5 1 3 4	1 3 5	5 1 3 5	1 4 5	5 3 4 5	-

Решење

Опишимо поступак којим од дате комбинације можемо добити следећу комбинацију у лексикографском редоследу. Поново тражимо *преломну тачку* тј. елемент који се може увећати. Пошто су комбинације дужине k и организоване су строго растуће, максимална вредност на последњој позицији је n, на претпоследњој $n-1$ итд. Дакле, последњи елемент се може увећати ако није једнак n, претпоследњи ако није једнак $n-1$ итд. Преломна тачка је позиција првог елемента који је мањи од свог максимума. Ако позиције бројимо од 0, максимум на позицији $k-1$ је n, на позицији $k-2$ је $n-1$ итд. тако да је максимум на позицији i једнак $n-k+1+i$. Ако преломна тачка не постоји (ако су све вредности на својим максимумима), наредна комбинација у лексикографском редоследу не постоји. У супротном увећавамо елемент на преломној позицији и да бисмо након тога добили лексикографски што мању комбинацију, све елементе иза њега постављамо на најмање могуће вредности. Пошто комбинација мора бити сортирана строго растуће, након увећања преломне вредности све елементе иза ње постављамо на вредност која је за један већа од вредности њој претходне вредности у низу. На пример, ако је $k = 6$, тада је наредна комбинација комбинацији 1256, комбинација 1345 - преломна вредност је 2 и она се може увећати на 3, након чега слажемо редом елементе за по један веће.

```
#include <iostream>
#include <vector>

using namespace std;

bool sledecaKombinacija(int n, vector<int>& kombinacija) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // krećemo od kraja i tražimo prvu poziciju koja nije na maksimumu
  // tj. koja se može povećati. Maksimumi od kraja su n, n-1, n-2, ...
  int i;
  for (i = k-1; i >= 0 && kombinacija[i] == n; i--, n--)
    ;
  // ako takva pozicija ne postoji, tekuća kombinacija je maksimalna
```

```
  if (i < 0)
    return false;
  // uvećavamo poslednji element koji se može povećati
  kombinacija[i]++;
  // iza njega slažemo redom brojeve za jedan veće
  for (i++; i < k; i++)
    kombinacija[i] = kombinacija[i-1] + 1;
  return true;
}


int main() {
  int n;
  cin >> n;
  vector<int> kombinacija;
  int x;
  while (cin >> x)
    kombinacija.push_back(x);
  if (sledecaKombinacija(n, kombinacija)) {
    for (int x : kombinacija)
      cout << x << " ";
    cout << endl;
  } else
    cout << "-" << endl;
  return 0;
}
```

Задатак: Све комбинације

Комбинације дужине k од n елемената подразумевају да се врши одабир k елемената скупа $\{1, \ldots, n\}$, слично као што се, на пример, у игри лото бира 7 од 39 куглица. Напиши програм који за дате вредности k и n набраја и исписује све комбинације, поређане по лексикографском редоследу.

Улаз: Прва линија стандардног улаза садржи број k ($1 \leq k \leq n$), а наредна број n ($2 \leq n \leq 20$).

Излаз: На стандардни излаз исписати све комбинације. Свака комбинација треба да буде представљена низом бројева сортираним строго растуће, а све комбинације треба да буду поређане у лексикографском редоследу.

Пример

Улаз	*Излаз*
3	1 2 3
5	1 2 4
	1 2 5
	1 3 4
	1 3 5
	1 4 5
	2 3 4
	2 3 5
	2 4 5
	3 4 5

Решење

Рекурзивни позиви по позицијама

Задатак рекурзивне функције биће да допуни низ дужине k од позиције i па до краја. Када је $i = k$, низ је попуњен и потребно је обрадити добијену комбинацију. У супротном бирамо елемент који ћемо поставити на позицију i. Пошто су комбинације уређене строго растуће, он мора бити већи од претходног (ако претходни не постоји, онда може бити 1) и мањи или једнак n. Заправо, ово горње ограничење мора да се смањи. Пошто

су елементи строго растући, а од позиције i па до краја низа треба поставити $k - i$ елемената, на позицији i може бити $n + i - k + 1$ и тада ће на позицији $k - 1$ бити вредност n. У петљи стављамо један по један од тих елемената на позицију i и рекурзивно настављамо генерисање од наредне позиције.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

// niz kombinacije dužine k na pozicijama [0, i) sadrži uređen
// niz elemenata iz skupa [1, n-i+1). Procedura na sve moguće
// načine dopunjava elementima iz skupa [1, n) tako da niz bude
// uređen rastući
void obradiSveKombinacije(vector<int>& kombinacija, int i, int n) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }
  // određujemo raspon elemenata na poziciji i
  int pocetak = i == 0 ? 1 : kombinacija[i-1]+1;
  int kraj = n + i - k + 1;
  // jedan po jedan element upisujemo na poziciju i, pa
  // nastavljamo generisanje rekurzivno
  for (int x = pocetak; x <= kraj; x++) {
    kombinacija[i] = x;
    obradiSveKombinacije(kombinacija, i+1, n);
  }
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacije(kombinacija, 0, n);
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

Рекурзивни позиви по вредностима

Постоји начин да избегнемо рекурзивне позиве у петљи. Током рекурзије можемо да чувамо информацију о томе који је распон елемената којим се проширује низ. Знамо да су то елементи скупа $\{1, \ldots, n\}$, међутим, пошто су комбинације сортиране растуће скуп кандидата је ужи. У претходном програму смо најмању

вредност за позицију i одређивали на основу вредности са позиције $i-1$, међутим, алтернативно можемо и експлицитно да одржавамо промењиве n_{min} и n_{max} које одређују скуп $\{n_{min}, \ldots, n_{max}\}$ чији се елементи распоређују у комбинацији на позицијама из интервала $[i, k)$. Ако је тај интервал празан, комбинација је попуњена и може се обрадити. У супротном, ако је $n_{min} > n_{max}$, тада не постоји вредност коју је могуће ставити на позицију i, па можемо изаћи из рекурзије, јер се тренутна комбинација не може попунити до краја. У супротном можемо размотрити две могућности. Прво на позицију i можемо поставити елемент n_{min} и рекурзивно извршити попуњавање низа од позиције $i+1$, а друго можемо тај елемент прескочити и у рекурзивном позиву поново захтевати да се попуни позиција i. У оба случаја се скуп елемената сужава на $\{n_{min}+1, \ldots, n_{max}\}$.

Претрагу можемо сасећи и мало раније. Наиме, пошто су понављања забрањена када је број елемената тог скупа (а то је $n - n_{min} + 1$) мањи од броја преосталих позиција које треба попунити (а то је $k - i$), већ тада можемо сасећи претрагу, јер не постоји могућност да се комбинација успешно допуни до краја.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

void obradiSveKombinacije(vector<int>& kombinacija, int i,
                          int n_min, int n_max) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }

  // ako tekuću kombinaciju nije moguće popuniti do kraja
  // prekidamo ovaj pokušaj
  if (n_max - n_min + 1 < k - i)
    return;

  // vrednost n_min uključujemo na poziciju i, pa rekurzivno
  // proširujemo tako dobijenu kombinaciju
  kombinacija[i] = n_min;
  obradiSveKombinacije(kombinacija, i+1, n_min+1, n_max);
  // vrednost n_min preskačemo i isključujemo iz kombinacije
  obradiSveKombinacije(kombinacija, i, n_min+1, n_max);
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacije(kombinacija, 0, 1, n);
}

int main() {
  int k, n;
```

```
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

Лексикографски следећа комбинација

Један начин да се задатак реши без рекурзије је да се употреби функција за одређивање наредне комбинације у лексикографском поретку која је описана у задатку Следећа комбинација.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

// pronalazi sledecu kombinaciju u leksikografskom redosledu
bool sledecaKombinacija(int n, vector<int>& kombinacija) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // krećemo od kraja i tražimo prvu poziciju koja nije na maksimumu
  // tj. koja se može povećati. Maksimumi od kraja su n, n-1, n-2, ...
  int i;
  for (i = k-1; i >= 0 && kombinacija[i] == n; i--, n--)
    ;
  // ako takva pozicija ne postoji, tekuća kombinacija je maksimalna
  if (i < 0)
    return false;
  // uvećavamo poslednji element koji se može povećati
  kombinacija[i]++;
  // iza njega slažemo redom brojeve za jedan veće
  for (i++; i < k; i++)
    kombinacija[i] = kombinacija[i-1] + 1;
  return true;
}


// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  // krecemo od kombinacije 1, 2, ..., k
  vector<int> kombinacija(k);
  for (int i = 0; i < k; i++)
    kombinacija[i] = i + 1;

  // obradjujemo kombinacije dokle god postoji sledeca
  do {
    obradi(kombinacija);
  } while (sledecaKombinacija(n, kombinacija));
}

int main() {
```

```
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

Лексикографски следећа комбинација - оптимизовано решење

Постоји и мала оптимизација претходног поступка. Наиме, ако знамо вредност преломне тачке у једном кораку, без поновне претраге можемо одредити вредност преломне тачке у наредном кораку. Кључно питање је то да ли након увећања преломне вредности она достиже свој максимум.

- Ако достиже тј. ако после увећања важи $komb_i = n-k+i+1$, тада после увећања преломне вредности, редом иза преломне тачке морају наћи елементи који су сви на својим максимумима, међутим, то је већ случај тако да није потребно поново их ажурирати. Наредна преломна тачка је непосредно испред текуће преломне тачке. На пример, наредна комбинација за 1356 је 1456. Преломна тачка је $i = 1$, важи да је $komb_1 = 3 = 6 - 4 + 1$ и довољно је само увећати елемент 3 на 4. Пошто су елементи од 4, 5 и 6, на својој максималној вредности, знамо да је наредна преломна вредност 1, па је наредна комбинација 2345.
- Ако након увећања преломна вредност она не достиже свој максимум тј. ако након увећања важи $komb_i < n - k + i + 1$, онда је након увећања и попуњавања низа до краја последњи елемент сигурно испод своје максималне вредности, тако да је наредна преломна тачка последња позиција у низу.

Интересантно, ова "оптимизација" не доноси никакву значајну добит и нема праткичних импликација. Ако обрада укључује било какву нетривијалну операцију или испис комбинације на екран, обрада потпуно доминира временом генерисања. Ако је обрада тривијална (на пример, само увећање глобалног бројача за један) тада не постоји значајна разлика у времену извршавања. Разлог томе је то што је у већини случајева преломна тачка последњи елемент или је врло близу десног краја, па је линеарна претрага брза.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
     // krecemo od kombinacije 1, 2, ..., k
   vector<int> kombinacija(k);
   for (int i = 0; i < k; i++)
     kombinacija[i] = i + 1;

   // obrađujemo prvu kombinaciju
   obradi(kombinacija);

   // specijalno obrađujemo slučaj n = k kada
   // nema drugih kombinacija
   if (n == k) return;

   // prva prelomna tačka je poslednja pozicija u nizu
   int i = k-1;
   while (i >= 0) {
      // ažuriramo kombinaciju
```

```
      kombinacija[i]++;
      // ako je uvećani prelomni element dostigao maksimum
      if (kombinacija[i] == n - k + i + 1)
        // naredna prelomna tačka je neposredno pre njega
        i--;
      else {
        // popunjavamo niz do kraja
        for (int j = i+1; j < k; j++)
           kombinacija[j] = kombinacija[j-1] + 1;
        // naredna prelomna tačka je poslednji element niza
        i = k-1;
     }
     // obrađujemo dobijenu kombinaciju
     obradi(kombinacija);
   }
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

Задатак: Све комбинације са понављањем

Из лото бубња у ком се налазе лоптице нумерисане бројевима од 1 до n вади се k лоптица, али се након што се запише број сваке извађене лоптице лоптица враћа у бубањ. Извучених k бројева се приказују у неопадајућем редоследу и представљају једну *комбинацију са понављањем*. Напиши програм који исписује све комбинације са понављањем за дате бројеве n и k.

Улаз: Са стандардног улаза се учитава број k ($2 \leq k \leq 8$) и број n ($2 \leq n \leq 8$) (сваки у посебној линији).

Излаз: На стандардни излаз исписати све комбинације са понављањем k елемената скупа $\{1, \ldots, n\}$, исписане у лексикографском редоследу.

Пример

Улаз	*Излаз*
2	1 1
3	1 2
	1 3
	2 2
	2 3
	3 3

Решење

Решење се може добити крајње једноставним прилагођавањем решења којим се генеришу све комбинације без понављања приказаних у задатку Све комбинације. Комбинације са понављањем се представљају неопадајућим (уместо строго растућим) низовима. Размотримо, на пример, рекурзивно решење. Поново вршимо два рекурзивна позива. Један када се на позицију i стави вредност n_{min} и други када се та вредност прескочи. Једина разлика је што се приликом постављања n_{min} на позицији i она оставља као минимална вредност и у рекурзивном позиву који се врши, јер сада не смета да се она понови – разлика је дакле само у прескакању увећавања једног јединог бројача.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
```

```
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}


void obradiSveKombinacijeSaPonavljanjem(vector<int>& kombinacija, int i,
                                        int n_min, int n_max) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }

  // ako ne postoji element koji možemo upisati na poziciju i,
  // kombinacija se ne može produžiti
  if (n_min > n_max)
    return;

  // vrednost n_min uključujemo na poziciju i, pa rekurzivno
  // proširujemo tako dobijenu kombinaciju
  kombinacija[i] = n_min;
  obradiSveKombinacijeSaPonavljanjem(kombinacija, i+1, n_min, n_max);
  // vrednost n_min preskačemo i isključujemo iz kombinacije
  obradiSveKombinacijeSaPonavljanjem(kombinacija, i, n_min+1, n_max);
}

void obradiSveKombinacijeSaPonavljanjem(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacijeSaPonavljanjem(kombinacija, 0, 1, n);
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacijeSaPonavljanjem(k, n);
  return 0;
}
```

Задатак: Следећа пермутација

Све пермутације бројева од 1 до n се могу поређати лексикографски. На пример за $n = 3$ пермутације у лексикографском поретку су

```
123
132
213
231
312
321
```

Написати програм којим се за дати природан број n и дату пермутацију бројева од 1 до n приказује следећа пермутација у лексикографском поретку (прва пермутација која се налази после дате пермутације).

Улаз: Прва линија стандардног улаза садржи природан број n ($n < 1000$). У свакој од n наредних линија

стандардног улаза, налазе се редом елементи пермутације, сваки у посебној линији.

Излаз: На стандардном излазу приказати редом елементе следеће пермутације у лексикографском поретку, сваки елемент у посебној линији. Ако не постоји следећа пермутација (дата пермутација је последња) приказати у једној линији поруку `ne postoji`.

Пример 1

Улаз	*Излаз*
5	3
3	1
1	5
4	2
5	4
2	

Пример 2

Улаз	*Излаз*
3	ne postoji
3	
2	
1	

Решење

Размотримо пермутацију 13542. Заменом елемента 2 и 4 би се добила пермутација 13524 која је лексикографски мања од полазне и то нам не одговара. Слично би се десило и да се замене елементи 5 и 4. Чињеница да је низ 542 строго опадајући нам говори да није могући ни на који начин разменити та три елемента да се добије лексикографски већа пермутација, тј. да је ово највећа пермутација која почиње са 13. Дакле, наредна пермутација ће бити лексикографски најмања пермутација која почиње са 14, а то је 14235.

Дакле, у првом кораку алгоритма проналазимо прву позицију i сдесна, такву да је $a_i < a_{i+1}$ (за све $i+1 \leq k < n-1$ важи да је $a_k > a_{k+1}$). Ово радимо најобичнијом линеарном претрагом (као у задатку Негативан број). У нашем примеру $a_i = 3$. Ако таква позиција не постоји, наша пермутација је скроз опадајућа и самим тим лексикографски највећа. Након тога, проналазимо прву позицију j сдесна такву да је $a_i < a_j$ (опет линеарном претрагом) и размењујемо елементе на позицијама i и j. У нашем примеру $a_j = 4$ и након размене добијамо пермутацију 14532. Пошто је овом разменом реп иза позиције i и даље стриктно опадајући, да бисмо добили жељену лексикографски најмању пермутацију која почиње са 14, потребно је обрнути редослед елемената репа (то можемо учинити као у задатку Обртање низа).

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool sledecaPermutacija(vector<int>& a){
  int n = a.size();

  // linearnom pretragom pronalazimo prvu poziciju i takvu da
  // je a[i] > a[i+1]
  int i = n - 2;
  while (i >= 0 && a[i] > a[i+1])
    i--;
  // ako takve pozicije nema, permutacija a je leksikografski maksimalna
  if (i < 0) return false;
  // linearnom pretragom pronalazimo prvu poziciju j takvu da
  // je a[j] > a[i]
  int j = n - 1;
  while (a[j] < a[i])
    j--;
  // razmenjujemo elemente na pozicijama i i j
  swap(a[i], a[j]);
  // obrcjemo deo niza od pozicije i+1 do kraja
  for (j = n - 1, i++; i < j; i++, j--)
    swap(a[i], a[j]);
  return true;
}

// ispisujemo elemente vektora
```

```
void prikazi(const vector<int>& a) {
  for (int x: a)
    cout << x << endl;
}

int main() {
  // ucitavamo polaznu permutaciju
  int n;
  cin >> n;
  vector<int> a(n);
  for(int i = 0; i < n; i++)
    cin >> a[i];
  // odredjujemo sledecu i ispisujemo rezultat
  if (sledecaPermutacija(a))
    prikazi(a);
  else
    cout << "ne postoji" << endl;
  return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// ispisujemo elemente vektora
void prikazi(const vector<int>& a) {
  for (int x: a)
    cout << x << endl;
}

int main() {
  // ucitavamo polaznu permutaciju
  int n;
  cin >> n;
  vector<int> a(n);
  for(int i = 0; i < n; i++)
    cin >> a[i];
  // odredjujemo sledecu i ispisujemo rezultat
  if (next_permutation(begin(a), end(a)))
    prikazi(a);
  else
    cout << "ne postoji" << endl;
  return 0;
}
```

Задатак: Све пермутације

Напиши програм који генерише и исписује све пермутације скупа $\{1, 2, \ldots, n\}$.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 8$).

Излаз: На стандардни излаз исписати тражене пермутације. Сваку пермутацију исписати у посебном реду, а елементе раздвојити по једним размаком. Редослед пермутација може бити произвољан.

Пример

Улаз	*Излаз*
3	1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1

Решење

Рекурзивно генерисање пермутација

Рекурзивно генерисање пермутација у лексикографском редоследу је веома компликовано, тако да ћемо се одрећи услова да пермутације морају бити поређане лексикографски.

У том случају можемо поступити на следећи начин. На прву позицију у низу у којем чувамо текућу пермутацију треба да постављамо један по један елемент скупа, а затим да рекурзивно одређујемо све пермутације преосталих елемената. Фиксиране елементе и елементе које треба пермутовати можемо чувати у истом низу. Нека на позицијама $[0, k)$ чувамо елементе које треба пермутовати, а на позицијама $[k, n)$ чувамо фиксиране елементе. Разматрамо позицију $k - 1$. Ако је $k = 1$, тада постоји само једна пермутација једночланог низа на позицији 0, њу придружујемо фиксираним елементима (пошто је она већ на месту 0 нема потребе ништа додатно радити) и исписујемо пермутацију. Ако је $k > 1$, тада је ситуација компликованија. Један по један елемент дела низа са позиција $[0, k)$ треба да доводимо на место $k-1$ и да рекурзивно позивамо пермутовање дела низа на позицијама $[0, k-1)$. Идеја која се природно јавља је да вршимо размену елемента на позицији $k - 1$ редом са свим елементима из интервала $[0, k)$ и да након сваке размене вршимо рекурзивне позиве. На пример, ако је низ на почетку 123, онда мењамо елемент 3 са елементом 1, добијамо 321 и позивамо рекурзивно генерисање пермутација низа 32 са фиксираним елементом 1 на крају. Затим у почетном низу мењамо елемент 3 са елементом 2, добијамо 132 и позивамо рекурзивно генерисање пермутација низа 13 са фиксираним елементом 2 на крају. Затим у почетном низу мењамо елемент 3 са самим собом, добијамо 123 и позивамо рекурзивно генерисање пермутација низа 12 са фиксираним елементом 3 на крају. Међутим, са тим приступом може бити проблема. Наиме, да бисмо били сигурни да ће на последњу позицију стизати сви елементи низа, размене морамо да вршимо у односу на *почетно* стање низа. Један начин је да се пре сваког рекурзивног позива прави копија низа, али постоји и ефикасније решење. Наиме, можемо као инваријанту функције наметнути да је након сваког рекурзивног позива распоред елемената у низу исти као пре позива функције. Уједно то треба да буде и инваријанта петље у којој се врше размене. На уласку у петљу распоред елемената у низу биће исти као на уласку у функцију. Вршимо прву размену, рекурзивно позивамо функцију и на основу инваријанте рекурзивне функције знамо да ће распоред након рекурзивног позива бити исти као пре њега. Да бисмо одржали инваријанту петље, потребно је низ вратити у почетно стање. Међутим, знамо да је низ промењен само једном разменом, тако да је довољно урадити исту ту размену и низ ће бити враћен у почетно стање. Тиме је инваријанта петље очувана и може се прећи на следећу позицију. Када се петља заврши, на основу инваријанте петље знаћемо да је низ исти као на улазу у функцију. На основу тога знамо и да ће инваријанта функције бити одржана и није потребно урадити ништа додатно након петље.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje permutaciju na standarndi izlaz
void obradi(const vector<int>& permutacija) {
  for (int x : permutacija)
    cout << x << " ";
  cout << endl;
}

void obradiSvePermutacije(vector<int>& permutacija, int k) {
  if (k == 1)
    obradi(permutacija);
  else {
```

```
    for (int i = 0; i < k; i++) {
      swap(permutacija[i], permutacija[k-1]);
      obradiSvePermutacije(permutacija, k-1);
      swap(permutacija[i], permutacija[k-1]);
    }
  }
}

void obradiSvePermutacije(int n) {
  vector<int> permutacija(n);
  for (int i = 1; i <= n; i++)
    permutacija[i-1] = i;
  obradiSvePermutacije(permutacija, n);
}

int main() {
  int n;
  cin >> n;
  obradiSvePermutacije(n);
  return 0;
}
```

Одређивање следеће permутације

Све permутације у лексикографском редоследу се могу добити тако што се крене од почетне permутације $1, 2, \ldots, n$ и у сваком кораку се исписује текућа permутација и мења се са наредном permутацијом у лексикографском поретку.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool sledecaPermutacija(vector<int>& a) {
  int n = a.size();

  // linearnom pretragom pronalazimo prvu poziciju i takvu da
  // je a[i] > a[i+1]
  int i = n - 2;
  while (i >= 0 && a[i] > a[i+1])
    i--;
  // ako takve pozicije nema, permutacija a je leksikografski maksimalna
  if (i < 0) return false;
  // linearnom pretragom pronalazimo prvu poziciju j takvu da
  // je a[j] > a[i]
  int j = n - 1;
  while (a[j] < a[i])
    j--;
  // razmenjujemo elemente na pozicijama i i j
  swap(a[i], a[j]);
  // obrcjemo deo niza od pozicije i+1 do kraja
  for (j = n - 1, i++; i < j; i++, j--)
    swap(a[i], a[j]);
  return true;
}

void obradi(const vector<int>& permutacija) {
  for (int x : permutacija)
```

```
    cout << x << " ";
  cout << endl;
}

void obradiSvePermutacije(int n) {
  vector<int> permutacija(n);
  // pocetna permutacija
  for (int i = 0; i < n; i++)
    permutacija[i] = i + 1;
  // prelazimo na narednu permutaciju, sve dok ih ima
  do {
    obradi(permutacija);
  } while (sledecaPermutacija(permutacija));
}

int main() {
  int n;
  cin >> n;
  obradiSvePermutacije(n);
}
```

Библиотечка функција за следећу пермутацију

У језику C++ функција `next_permutation` декларисана у заглављу `<algorithm>` одређује наредну пермутацију у односу на дату. Функцији се прослеђују два итератора који ограничавају распон елемената у којима се налази пермутација.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> permutacija(n);
  iota(begin(permutacija), end(permutacija), 1);
  do {
    for (int x : permutacija)
      cout << x << " ";
    cout << endl;
  } while (next_permutation(begin(permutacija), end(permutacija)));
}
```

Задатак: Сви палиндроми од дате речи

Дата је реч r написана малим словима енглеске абецеде. Написати програм којим се приказују у лексикографском поретку сви палиндроми који се могу добити размештањем слова дате речи.

Улаз: Прва и једина линија стандардног улаза садржи реч r са највише 20 малих слова енглеске абецеде.

Излаз: На стандардном излазу приказати тражене палиндроме, ако се не може формирати ни један палиндром приказати цртицу `-`.

Пример 1

Улаз	*Излаз*
racecar	acrerca
	arcecra
	carerac
	craearc
	racecar
	rcaeacr

Пример 2

Улаз	*Излаз*
abcdecda	-

Решење

Задатак можемо решити тако што прво приметимо да се од датих слова може формирати палиндром ако и само се сва слова у речи јављају паран број пута (осим у случају речи непарне дужине у којој се средишње слово јавља непаран број пута). Дакле, пребројаћемо појављивања свих слова у речи (помоћу низа бројача или мапе тј. речника), затим ћемо пронаћи слова која се јављају непаран број пута. Палиндром ће бити могуће направити ако и само ако је реч парне дужине и нема слова која се појављују непаран број пута или је реч непарне дужине и постоји тачно једно слово које се појављује непаран број пута.

Ако се од речи може направити палиндром, тада њена слова можемо поделити на две половине (када се евентуално изузме средишње слово, сва се слова јављају паран број пута и једну половину њихових појављивања ћемо поставити у леви, а другу у десни део палиндрома). Било која пермутација слова леве половине речи даје неки палиндром - само је потребно да слова у десној половини речи буду распоређена у обратном редоследу. Стога крећемо од најмање пермутације у лексикографском редоследу (то је она у којој су слова сортирана) у левој половини речи и затим формирамо један по један палиндром одређујући следећу пермутацију, све док оне постоје. Алгоритам за одређивање лексикографски следеће пермутације описан је у задатку Следећа пермутација.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <map>

using namespace std;

// odredjuje broj pojavljivanja svakog slova u datoj reci
map<char, int> prebrojSlova(const string& r) {
  map<char, int> broj;
  for (char c : r)
    broj[c]++;
  return broj;
}

// odredjuje slova koja se javljaju neparan broj puta
vector<char> neparnaSlova(const map<char, int>& broj) {
  vector<char> rez;
  for (auto it : broj)
    if (it.second % 2 == 1)
      rez.push_back(it.first);
  return rez;
}

// odredjuje leksikografski sledecu permutaciju prvih n karaktera reci r
bool sledecaPermutacija(string& r, int n) {
  int i, j;
  i = n - 1;
  while (i > 0 && r[i] <= r[i - 1])
    i--;
  if (i == 0)
    return false;
  j = n - 1;
  while (r[j] <= r[i - 1])
```

```
    j--;
  swap(r[i - 1], r[j]);
  for (j = n - 1; i < j; i++, j--)
    swap(r[i], r[j]);
  return true;
}

// ispisuje sve palindrome koji se mogu dobiti permutovanjem slova date reci r
void sviPalindromi(const string& r) {
  map<char, int> broj = prebrojSlova(r);
  vector<char> neparna = neparnaSlova(broj);

  // sva slova u palindromu se javljaju paran broj puta, osim
  // eventualno sredisnjeg koje se moze javiti neparan broj puta
  bool moze =
    (neparna.size() == 0 && r.length() % 2 == 0) ||
    (neparna.size() == 1 && r.length() % 2 == 1);

  if (!moze) {
    cout << "-" << endl;
    return;
  }

  // rezultujuci palindrom
  string palindrom(r.length(), ' ');
  // popunjavamo prvu polovinu tako da bude sortirana
  int i = 0;
  for (auto it : broj)
    for (int j = 0; j < it.second / 2; j++)
      palindrom[i++] = it.first;

  // duzina polovine
  int d = i;

  // postavljamo sredisnje slovo ako ono postoji
  if (neparna.size() > 0)
    palindrom[i++] = neparna[0];

  // pozicija pocetka druge polovine
  int p = i;


  // odredjujemo sve permutacije prve polovine reci
  do {
    // kopiramo prvu polovinu palindroma u drugu
    copy(begin(palindrom), next(begin(palindrom), d), next(begin(palindrom), p));
    // obrcemo drugu polovinu palindroma
    reverse(next(begin(palindrom), p), end(palindrom));
    // ispisujemo tekuci palindrom
    cout << palindrom << endl;
  } while(sledecaPermutacija(palindrom, d));
}

int main() {
   string r;
   cin >> r;
   sviPalindromi(r);
   return 0;
```

```
}
```

Лексикографски следећу пермутацију је могуће одредити и применом библиотечке функције next_permutation.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <map>

using namespace std;

// odredjuje broj pojavljivanja svakog slova u datoj reci
map<char, int> prebrojSlova(const string& r) {
  map<char, int> broj;
  for (char c : r)
    broj[c]++;
  return broj;
}

// odredjuje slova koja se javljaju neparan broj puta
vector<char> neparnaSlova(const map<char, int>& broj) {
  vector<char> rez;
  for (auto it : broj)
    if (it.second % 2 == 1)
      rez.push_back(it.first);
  return rez;
}

void sviPalindromi(string r) {
  map<char, int> broj = prebrojSlova(r);
  vector<char> neparna = neparnaSlova(broj);

  // sva slova u palindromu se javljaju paran broj puta, osim
  // eventualno sredisnjeg koje se moze javiti neparan broj puta
  bool moze =
    (neparna.size() == 0 && r.length() % 2 == 0) ||
    (neparna.size() == 1 && r.length() % 2 == 1);

  if (!moze) {
    cout << "-" << endl;
    return;
  }

  // rezultujuci palindrom
  string palindrom(r.length(), ' ');
  // popunjavamo prvu polovinu tako da bude sortirana
  int i = 0;
  for (auto it : broj)
    for (int j = 0; j < it.second / 2; j++)
      palindrom[i++] = it.first;

  // duzina polovine
  int d = i;

  // postavljamo sredisnje slovo ako ono postoji
  if (neparna.size() > 0)
    palindrom[i++] = neparna[0];
```

```
  // pozicija pocetka druge polovine
  int p = i;


  // odredjujemo sve permutacije prve polovine reci
  do {
    // kopiramo prvu polovinu palindroma u drugu
    copy(begin(palindrom), next(begin(palindrom), d), next(begin(palindrom), p));
    // obrcemo drugu polovinu palindroma
    reverse(next(begin(palindrom), p), end(palindrom));
    // ispisujemo tekuci palindrom
    cout << palindrom << endl;
  } while(next_permutation(begin(palindrom), next(begin(palindrom), d)));
}

int main() {
   string r;
   cin >> r;
   sviPalindromi(r);
   return 0;
}
```

Палиндроме можемо генерисати и посебно дизајнираном рекурзивном функцијом. Ако знамо скуп слова која треба распоредити ван евентуалне средишње позиције, можемо редом анализирати могућности за прво (а уједно и последње слово). Пошто палиндроми треба да буду генерисани у лексикографском редоследу, на прво место ћемо стављати слова из тог скупа, редом, у абецедном редоследу. Након постављања неког слова на прво и последње место, изузећемо га из скупа и рекурзивно ћемо наставити попуњавање унутрашњости палиндрома. Излаз из рекурзије је када се скуп слова испразни (тада је реч попуњена и можемо је исписати).

```
#include <iostream>
#include <map>
#include <vector>
using namespace std;

// odredjuje broj pojavljivanja svakog slova u datoj reci
map<char, int> prebrojSlova(const string& r) {
  map<char, int> broj;
  for (char c : r)
    broj[c]++;
  return broj;
}

// odredjuje slova koja se javljaju neparan broj puta
vector<char> neparnaSlova(const map<char, int>& broj) {
  vector<char> rez;
  for (auto it : broj)
    if (it.second % 2 == 1)
      rez.push_back(it.first);
  return rez;
}

// na sve moguce nacine popunjava nisku s slovima ciji je broj pojavljivanja odredjen
// mapom brojSlova - pretpostavlja se da je prvih i poslednjih k slova niske s vec
// popunjeno tako da s u tom delu zadovoljava uslov palindroma, kao i da je eventualno
// sredisnje slovo niske s vec popunjeno
void sviPalindromi(string& s, int k, map<char, int>& brojSlova) {
  // ako je cela niska popunjena, ispisujemo je
  if (k == s.length() / 2)
    cout << s << endl;
```

```
  // analiziramo sva slova iz mape
  for (auto it: brojSlova) {
    // svako slovo koje se javlja bar dva puta
    if (it.second >= 2) {
      // rasporedjujemo na pocetak i na kraj tekuceg sredisnjeg dela reci
      s[k] = s[s.length() - 1 - k] = it.first;
      // uklanjamo ta dva pojavljivanja iz skupa slova
      brojSlova[it.first] -= 2;
      // rekurzivno prelazimo na popunjavanje sredisnjeg dela bez
      // postavljena dva krajnja slova
      sviPalindromi(s, k+1, brojSlova);
      // vracamo ta dva pojavljivanja iz skupa slova
      brojSlova[it.first] += 2;
    }
  }
}

// ispisuje sve palindrome koji se mogu dobiti permutovanjem slova date reci s
void sviPalindromi(const string& s) {
  map<char, int> brojSlova = prebrojSlova(s);
  vector<char> neparna = neparnaSlova(brojSlova);

  // sva slova u palindromu se javljaju paran broj puta, osim
  // eventualno sredisnjeg koje se moze javiti neparan broj puta
  bool moze =
    (neparna.size() == 0 && s.length() % 2 == 0) ||
    (neparna.size() == 1 && s.length() % 2 == 1);

  if (!moze) {
    cout << "-" << endl;
    return;
  }

  // rezultujuci palindrom
  string palindrom(s.length(), ' ');

  // postavljamo sredisnje slovo ako ono postoji
  if (neparna.size() == 1)
    palindrom[palindrom.length() / 2] = neparna[0];

  // rekurzivnom funkcijom odredjujemo sve palindrome
  sviPalindromi(palindrom, 0, brojSlova);
}

int main() {
  string s;
  cin >> s;
  sviPalindromi(s);
  return 0;
}
```

Задатак: Следећа партиција

Партиције броја n представљају разлагање тог броја на сабирке чија је вредност између 1 и n. На пример, број 10 се може партиционисати као $5+2+2+1$. Свака партиција се може нормализовати тако што се претпостави, на пример, да су сабирци сортирани нерастуће. Напиши програм који за дату партицију одређује следећу партицију у лексикографском редоследу.

Улаз: Са стандардног улаза се уноси нормализована партиција при чему су сабирци раздвојени карактером + (њихов збир је мањи од 1000).

Излаз: На стандардни излаз исписати наредну нормализовану партицију у лексикографском редоследу (у једној линији, при чему су сабирци раздвојени карактером +) или реч `ne` ако таква партиција не постоји.

Пример

Улаз	*Излаз*
5+2+2+1	5+3+1+1

Решење

Да бисмо добили следећу партицију у лексикографском редоследу, потребно је увећати за један неки елемент који је што ближи крају низа, док префикс низа треба да остане непромењен. Ако је партиција једночлана, тада је она лексикографски највећа. Последњи елемент низа није могуће повећати за један, јер би због очувања збира елемената партиције неки елемент пре њега морао бити смањен. Наредни кандидат за повећање је претпоследњи елемент, а он се може увећати на један само ако се на месту испред њега не налази елемент који му је једнак, јер би се тада повећањем претпоследњег елемента добила партиција која није нормализована (уређена нерастући). У супротном разматрамо елемент пре претпоследњег и тако редом, све док не наиђемо на елемент испред којег не стоји елемент који му је једнак. Тај елемент повећавамо за 1. Након тога потребно је поправити елементе иза тог увећаног елемента, тако да партиција буде лексикографски што мања. То ће се десити ако се иза увећаног елемента поставе само јединице. Да се збир не би променио, број постављених јединица треба да буде за један мањи од збира свих елемената иза елемента који смо увећали за један (тај збир можемо израчунавати док обилазимо низ уназад тражећи најдешњи елемент који се може увећати за 1).

Пошто се дужина партиције може променити приликом преласка на следећу партицију, уместо класичног низа партицију можемо представити неким обликом низа који допушта додавање елемената на крај.

```
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

bool sledecaParticija(vector<int>& particija)
{
  int k = particija.size();
  // ako je tekuca particija jednoclana, ne postoji sledeca
  if (k == 1)
    return false;

  // pronalazimo poziciju prvog elementa zdesna koji se moze uvecati i
  // ujedno racunamo zbir elemenata iza njega
  int i;
  int zbir = particija[k-1];
  for (i = k-2; i > 0 && particija[i] == particija[i-1]; i--)
    zbir += particija[i];

  // uklanjamo sve elemente iza pozicije i
  particija.resize(i+1);

  // uvecavamo element koji smemo uvecati
  particija[i]++;

  // dodajemo jedinice do kraja particije
  for (int m = 0; m < zbir - 1; m++)
    particija.push_back(1);
  return true;
}
```

```
int main() {

  vector<int> particija;
  while (true) {
    int x;
    cin >> x;
    particija.push_back(x);
    char c = cin.get();
    if (c != '+')
      break;
  }
  if (sledecaParticija(particija)) {
    cout << particija[0];
    for (int i = 1; i < particija.size(); i++)
      cout << "+" << particija[i];
    cout << endl;
  } else
    cout << "ne" << endl;
  return 0;
}
```

Задатак: Све партиције

Партиције броја n представљају разлагање тог броја на сабирке чија је вредност између 1 и n. На пример, број 10 се може партиционисати као $5 + 2 + 2 + 1$. Свака партиција се може нормализовати тако што се претпостави, на пример, да су сабирци сортирани нерастуће. Напиши програм који исписује све партиције датог броја.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 25$).

Излаз: На стандардни излаз исписати све нормализоване партиције броја n, сортиране лексикографски растуће.

Пример

Улаз	*Излаз*
5	1 1 1 1 1
	2 1 1 1
	2 2 1
	3 1 1
	3 2
	4 1
	5

Решење

Свака партиција има свој први сабирак. Свакој партицији броја n којој је први сабирак s (при чему је $1 \leq s \leq n$) једнозначно одговара нека партиција броја $n - s$, што указује да се проблем може решавати индуктивно-рекурзивном конструкцијом. Пошто је сабирање комутативно, да не бисмо суштински исте партиције понављали више пута наметнућемо услов да сабирци у свакој партицији буду сортирани нерастуће. Дакле, ако је први сабирак s, сви сабирци иза њега морају да буду мањи или једнаки од s. Зато нам није довољно само да умемо да генеришемо све партиције броја $n - s$, већ је потребно да ојачамо индуктивну хипотезу. Претпоставићемо да се у датом вектору на позицијама $[0, i)$ налазе раније постављени елементи партиције и да је задатак процедуре да тај низ допуни на све могуће начине партицијама броја n у којима су сви сабирци мањи или једнаки s_{max}. Излаз из рекурзије представљаће случај $n = 0$ у ком је једина могућа партиција броја 0 празан скуп, у коме нема сабирака. Тада сматрамо да је партиција успешно формирана и обрађујемо садржај вектора.

Рекурзија по позицијама

Један начин да се партиције допуне је да се размотре све могуће варијанте за сабирак на позицији i. На основу услова они морају бити већи од нуле, мањи или једнаки s_{max}, а природно је да морају да буду и мањи или

једнаки од n (јер први сабирак не може бити већи од збира природних бројева). Ако је m мањи од бројева n и s_{max}, могући први сабирци су сви бројеви $1 \leq s' \leq m$. Када фиксирамо сабирак s' низ рекурзивно допуњавамо свим партицијама броја $n - s'$ у којима су сви сабирци мањи или једнаки s', јер је потребно преостали део збира представити као партицију бројева који нису већи од s'.

У главној функцији ћемо алоцирати низ дужине n (јер најдужа партиција има n сабирака који су сви једнаки 1) и захтеваћемо да се тај низ попуни почевши од позиције 0 партицијама броја n у којима су сви сабирци мањи или једнаки n.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void obradi(const vector<int>& particija, int k) {
  for (int i = 0; i < k; i++)
    cout << particija[i] << " ";
  cout << endl;
}

void obradiParticije(int n, int smax, vector<int>& particija, int k) {
  if (n <= 0)
    obradi(particija, k);
  else {
    for (int s = 1; s <= min(n, smax); s++) {
      particija[k] = s;
      obradiParticije(n-s, s, particija, k+1);
    }
  }
}

void obradiParticije(int n) {
  vector<int> particija(n);
  obradiParticije(n, n, particija, 0);
}

int main() {
  int n;
  cin >> n;
  obradiParticije(n);
  return 0;
}
```

Рекурзија по вредностима

Уместо да се анализирају све могуће вредности сабирка на позицији i, могуће је разматрати само две могућности: прву да се на позицији i јавља сабирак s_{max}, а другу да се на позицији i јавља неки сабирак строго мањи од s_{max}. Први случај је могућ само ако је $n \geq s_{max}$ и када се на позицију i постави s_{max} низ допуњујемо од позиције $i + 1$ партицијама броја $n - s_{max}$ у којима су сви сабирци мањи или једнаки s_{max}. Други случај је увек могућ и тада партицију допуњујемо партицијама броја n у којима је највећи сабирак $s_{max} - 1$. У зависности од редоследа ова два рекурзивна позива одређује се да ли ће пермутације бити сортиране лексикографски растуће или опадајуће.

```
#include <iostream>
#include <vector>

using namespace std;

void obradi(const vector<int>& particija, int k) {
```

```
  for (int i = 0; i < k; i++)
    cout << particija[i] << " ";
  cout << endl;
}

void obradiParticije(int n, int smax, vector<int>& particija, int k) {
  if (n == 0)
    obradi(particija, k);
  else {
    if (smax == 0) return;
    obradiParticije(n, smax-1, particija, k);
    if (smax <= n) {
      particija[k] = smax;
      obradiParticije(n-smax, smax, particija, k+1);
    }
  }
}

void obradiParticije(int n) {
  vector<int> particija(n);
  obradiParticije(n, n, particija, 0);
}

int main() {
  int n;
  cin >> n;
  obradiParticije(n);
  return 0;
}
```

Још један начин да се реши задатак је да се итеративно набрајају следеће партиције у лексикографском редоследу (кренувши од партиције која има n јединица) све док се не дође до последње партиције која садржи само број n. Проналажење следеће партиције у лексикографском редоследу могуће је урадити функцијом која је описана у задатку Следећа партиција.

```
#include <iostream>
#include <vector>

using namespace std;

void obradi(const vector<int>& particija, int k) {
  for (int i = 0; i < k; i++)
    cout << particija[i] << " ";
  cout << endl;
}

bool sledecaParticija(vector<int>& particija, int& k) {
  // ako je tekuca particija jednoclana, ne postoji sledeca
  if (k == 1)
    return false;

  // pronalazimo poziciju prvog elementa zdesna koji se moze uvecati i
  // ujedno racunamo zbir elemenata iza njega
  int i;
  int zbir = particija[k-1];
  for (i = k-2; i > 0 && particija[i] == particija[i-1]; i--)
    zbir += particija[i];
  // uvecavamo element koji smemo uvecati
  particija[i]++;
```

```
  // postavljamo jedinice do kraja particije
  for (k = i+1; --zbir > 0; k++)
    particija[k] = 1;
  return true;
}

void obradiParticije(int n) {
  vector<int> particija(n, 1);
  int k = n;
  do {
    obradi(particija, k);
  } while (sledecaParticija(particija, k));
}

int main() {
  int n;
  cin >> n;
  obradiParticije(n);
  return 0;
}
```

Задатак: Све једноцифрене партиције

Напиши програм који исписује све могуће начине да се дати број n напише као збир једноцифрених позитивних бројева.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 15$).

Излаз: На стандардни излаз исписати све тражене партиције, у лексикографском редоследу. Елементи сваке партиције се исписују у истом реду, са по једним размаком иза сваког елемента. Редослед елемената је битан, што значи да да партиције које се састоје од истих сабирака у различитом редоследу сматрамо различитим (треба их све исписати).

Пример

Улаз	*Излаз*
4	1 1 1 1
	1 1 2
	1 2 1
	1 3
	2 1 1
	2 2
	3 1
	4

Решење

Све партиције на једноцифрене бројеве можемо набројати рекурзивном процедуром. Партиције можемо памтити у низу (можемо алоцирати простор за смештање n елемената, где је n полазни број чије партиције разматрамо, јер се најдужа партиција састоји од n јединица. Уз број n који се раставља и низ у коме се чувају елементи партиције, прослеђиваћемо и број тренутно попуњених елемената тог низа. На текуће место у низу постављаћемо једну по једну цифру c и рекурзивно позивати процедуру чији ће задатак бити да на све могуће начине допуни тај низ партицијама броја n умањеног за цифру c тј. партицијама броја $n - c$. У случају када је број n једноцифрен, горња граница за цифру c биће број n, а не цифра 9. Излаз из рекурзије представља тренутак када n достигне вредност 0, што значи да је збир тренутно уписаних цифара у низу једнак полазном броју и тада се партиција исписује.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
```

```
void ispisi(vector<int> particija, int k) {
  for (int i = 0; i < k; i++)
    cout << particija[i] << " ";
  cout << endl;
}

void ispisiJednocifreneParticije(int n, vector<int>& particija, int k) {
  if (n == 0) {
    ispisi(particija, k);
  } else {
    for (int c = 1; c <= min(9, n); c++) {
      particija[k] = c;
      ispisiJednocifreneParticije(n-c, particija, k+1);
    }
  }
}

void ispisiJednocifreneParticije(int n) {
  vector<int> particija(n);
  ispisiJednocifreneParticije(n, particija, 0);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  ispisiJednocifreneParticije(n);
  return 0;
}
```

Задатак: Сви n-тоцифрени бројеви са датим збиром цифара

Написати програм који исписује све n-тоцифрене бројеве који имају дати збир цифара.

Улаз: Прва линија садржи збир k ($1 \leq k \leq 9n$), а друга број цифара n ($2 \leq n \leq 100$).

Излаз: На стандардни излаз исписати све тражене бројеве, уређене по величини.

Пример

Улаз	*Излаз*
24	699
3	789
	798
	879
	888
	897
	969
	978
	987
	996

Решење

Наивно решење у ком би се генерисали сви n-тоцифрени бројеви (слично као у задатку Све варијације, а затим филтрирали они чији је збир цифара једнак датом броју је прилично неефикасно.

Овај задатак је донекле сличан задатку Све једноцифрене партиције, па боље решење можемо засновати на рекурзивној процедури генерисања свих партиција приказаној у том задатку. Основна разлика у овом задатку је то што се у овом задатку захтева да је број елемената у свакој партицији једнак датом броју цифара. Зато партицију исписујемо тј. излаз из рекурзије вршимо само ако је њена тренутна дужина баш једнака том

броју и ако је преостала вредност параметра n постала једнака нули. Рекурзивни корак вршимо само ако је у партицији преостало још места тј. ако је број тренутно попуњених елемената строго мањи од задатог броја цифара. Још једна ситна разлика у односу на задатак Све једноцифрене партиције је то што је допуштено коришћење цифре нула (на свим позицијама, осим на почетној).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void ispisi(vector<int> particija, int k) {
  for (int i = 0; i < k; i++)
    cout << particija[i];
  cout << endl;
}

void ispisiJednocifreneParticije(int n, vector<int>& particija, int k) {
  if (n == 0 && k == particija.size())
    ispisi(particija, k);
  else if (k < particija.size())
    for (int c = k == 0 ? 1 : 0; c <= min(9, n); c++) {
      particija[k] = c;
      ispisiJednocifreneParticije(n-c, particija, k+1);
    }
}

void ispisiJednocifreneParticije(int n, int brojCifara) {
  vector<int> particija(brojCifara);
  ispisiJednocifreneParticije(n, particija, 0);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  int brojCifara;
  cin >> brojCifara;
  ispisiJednocifreneParticije(n, brojCifara);
  return 0;
}
```

Одсецање на основу доње границе за сваку цифру

У претходном решењу се на сваку позицију постављају све цифре од нуле (или евентуално 1, на почетној позицији), па до 9 (уз евентуално одсецање када је број n мањи од 9). Тако, на пример, може да се деси да приликом покушаја генерисања троцифрених бројева са збиром цифара 27 на месту прве цифре испробавамо вредности од 1 до 8, а да заправо ни са једном од њих не можемо да попунимо партицију до краја (јер је једино решење 999). Много ефикасније решење добијамо ако применимо још једно одсецање и одредимо доњу границу вредности текуће цифре. Наиме, максимални могући збир цифара иза текуће се лако добије као $9(m-1)$, где је m број тренутно непопуњених цифара у партицији. Ако је текућа цифра једнака c, тада мора да важи да је преостали збир цифара n мањи или једнак $c+9(m-1)$, одакле се добија граница да је $c \geq n-9(m-1)$. Дакле, у петљи која поставља текућу цифру крећемо од веће од вредности $c+9(m-1)$ и вредности 0 (тј. 1 на почетној позицији), а завршавамо са мањом од вредности n и 9. С обзиром на овако одређене границе, имамо гаранцију да ће свака партиција моћи успешно да се попуни и при изласку из рекурзије само треба да контролишемо да ли су све цифре партиције потпуно попуњене (ако јесу, тада ће вредност преосталог збира n сигурно бити једнака нули).

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>

using namespace std;

void ispisi(vector<int> particija) {
  for (int x : particija)
    cout << x;
  cout << endl;
}

void ispisiJednocifreneParticije(int n, vector<int>& particija, int k) {
  if (k == particija.size()) {
    ispisi(particija);
  } else {
    int preostaloCifara = particija.size() - k;
    int maksZbirIza = 9 * (preostaloCifara - 1);
    int minC = max(k == 0 ? 1 : 0, n - maksZbirIza);
    int maksC = min(9, n);
    for (int c = minC; c <= maksC; c++) {
      particija[k] = c;
      ispisiJednocifreneParticije(n-c, particija, k+1);
    }
  }
}

void ispisiJednocifreneParticije(int n, int brojCifara) {
  vector<int> particija(brojCifara);
  ispisiJednocifreneParticije(n, particija, 0);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  int brojCifara;
  cin >> brojCifara;
  ispisiJednocifreneParticije(n, brojCifara);
  return 0;
}
```

Задатак: K-та тешка реч

Реч је „лака“ ако садржи два суседна подниза истих слова (две узастопне исте подречи). У супротном, реч је „тешка“. Примери „лаких“ речи су `BBA` (подреч је `B`), `ABCDACABCABA` (подреч је `CAB`), `BACBAC` (подреч је `BAC`), а „тешких“ су `D`, `DC`, `CBABCBA`. Написати програм којим се за дати природан број n и природан број k одређује k-та „тешка“ реч по лексикографском поретку која се добија од првих n великих слова енглеске абецеде. Празна реч је тешка и она се сматра нултом речју.

Улаз: Прва линија стандардног улаза садржи природан број n ($2 \leq n \leq 26$), друга линија садржи природан број K ($1 \leq K \leq 10^5$).

Излаз: На стандардном излазу у једној линији приказати тражену реч. Ако тражена реч не постоји исписати поруку `ne postoji`.

Пример 1

Улаз	*Излаз*
2	BAB
6	

Пример 2

Улаз	*Излаз*
2	ne postoji
7	

Пример 3

Улаз	*Излаз*
3	ABACABCACBA
12	

Решење

Задатак можемо решити рекурзивном функцијом која набраја тешке речи у лексикографском редоследу. Ово се може постићи модификацијом алгоритма за генерисање варијација у лексикографском редоследу (алгоритам је донекле сличан оном приказаном у задатку Све варијације. Рекурзивна функција прима текућу тешку реч и покушава да је на све начине прошири генеришући нове тешке речи. Полазни позив се врши за празну реч (она је нулта), која се онда на све могуће начине проширује.

На основу дефиниције лексикографског поретка, речи које се добијају проширивањем полазне су иза ње у лексикографском поретку, а испред су речи које се добијају изменом завршних карактера полазне речи. Наша функција генерише тешке речи проширивањем дате полазне речи. Проширивање тече тако што се полазна реч проширује редом једним по једним допуштеним словом, при чему се за сваку тако добијену реч проверава да ли је тешка и ако јесте, она се даље проширује рекурзивним позивом. Ако је добијена реч лака, нема је смисла даље проширивати јер су све речи које се добију проширивањем лаке речи и даље лаке.

Проверу да ли је реч тешка можемо убрзати захваљујући томе што знамо да је текућа реч добијена тако што је нека тешка реч проширена једним словом. Знамо да полазна тешка реч нема поновљених узастопних подречи, па је једина могућност да постоји нека подреч која се понавља та да се неки суфикс проширене речи понавља. Иако постоје ефикаснији начини да се ово испита (коришћењем напреднијих алгоритама обраде текста, суфиксних стабала и слично), у нашој имплементацији ћемо проверу поновљених суфикса извршити грубом силом (за сваки суфикс који није дужи од половине дужине речи ћемо проверавати да ли се јавља непосредно испред себе). Иако је теоријски сложеност најгорег случаја ове провере квадратна (јер се проверава линеарни број суфикса, чија дужина током алгоритма расте линеарно), у пракси ће се десити да ће се различитост суфикса и ниске испред њега често установити доста брзо (већ после провере тек неколико слова), па се у пракси показује да ову проверу можемо сматрати практично линеарном.

Пошто дужина речи није ограничена, могуће је да се проширивање никада не заустави. Наш задатак је да генерисање нових тешких речи зауставимо када генеришемо k-ту реч по реду. Стога ћемо функцији прослеђивати и референтни параметар који ће се смањивати током рекурзије, сваки пут када се генерише нова тешка реч (а оне се по дизајну наше функције генеришу у лексикографском редоследу). Ако тај бројач некада достигне вредност 0, генерисано је k речи и наша функција ће тада вратити текућу тешку реч (та реч ће заправо бити смештена у ниску карактера која се мења током рекурзија, а функција ће вратити истинитосну вредност тачно чиме ће позиваоцима послати сигнал да генерисање речи треба да престане и да је k-та реч успешно пронађена). Ако се заврши са проширивањем речи свим могућим допуштеним карактерима и при том се не генерише тражених k речи, тада функција враћа логичку вредност нетачно, остављајући ниску у стању у ком је позвана (како би се на претходном нивоу рекурзије могло заменити последње слово и како би се наставило генерисање). Пошто се приликом генерисања сваке тешке речи бројач k умањује за 1, након рекурзивног позива бројач k ће бити умањен за укупан број речи које су генерисане проширивањем полазне речи за тај рекурзивни позив (тако да ће све време рада алгоритма важити да треба исписати k-ту тешку реч која лексикографски следи иза полазне).

```
#include <iostream>
#include <string>

using namespace std;

// provera da li se sufiks duzine d dva puta uzastopno ponavlja
bool ponovljenSufiks(const string& s, int d) {
  int n = s.length();
  for (int i = 1; i <= d; i++)
    if (s[n-i] != s[n-d-i])
      return false;
  return true;
}

// provera da li postoji neki pravi sufiks koji se dva puta uzastopno
// ponavlja
bool sufiksJeLak(const string& s) {
  int n = s.length();
  for (int d = 1; d <= n/2; d++)
    if (ponovljenSufiks(s, d))
      return true;
```

```
  return false;
}

// pronalazi k-tu tesku rec koja se moze dobiti prosirivanjem teske reci s
//  - ako uspe, funkcija vraca true i rezultat se nalazi u niski s
//  - ako ne uspe, funkcija vraca false, s je nepromenjen, a k je umanjen
//    za broj svih teskih reci koje su dobijene prosirivanjem niske s
bool ktaTeskaRec(int brSlova, string& s, int& k) {
  // nulta rec je upravo tekuca rec s
  if (k == 0)
    return true;
  // string s prosirujemo jednim po jednim mogucim slovom
  s.push_back(' ');
  int n = s.length();
  for (int i = 0; i < brSlova; i++) {
    s[n-1] = 'A' + i;
    // lake reci preskacemo
    if (!sufiksJeLak(s)) {
      // pronasli smo novu tesku rec, pa umanjujemo brojac
      k--;
      // nastavljamo prosirivanje tekuce reci - rekurzivni poziv ce
      // umanjivati k za svaku pronadjenu rec i ako k stigne do nule
      // (ako se stigne do k-te reci), vratice vrednost true
      if (ktaTeskaRec(brSlova, s, k))
        return true;
    }
  }
  // nismo uspeli da prosirimo rec do k-te, pa vracamo nisku u pocetno
  // stanje i vracamo false
  s.pop_back();
  return false;
}

int main() {
   int brSlova, k;
   cin >> brSlova >> k;
   // pokusavamo da pronadjemo k-tu tesku rec prosirivanjem prazne
   // reci k puta
   string s = "";
   if (ktaTeskaRec(brSlova, s, k))
     cout << s << endl;
   else
     cout << "ne postoji" << endl;
   return 0;
}
```

Задатак можемо решити и помоћу функције која на основу дате тешке речи проналази следећу тешку реч. То се може урадити донекле слично алгоритму одређивања следеће варијације у лексикографском редоследу (као у задатку Следећа варијација). За дату реч се проналази лексикографски следећа реч све док се не установи да је та реч тешка или док се не установи да не постоји наредна течка реч.

Полазна реч се проширује карактером A (добијена реч је лексикографски следбеник полазне). Ако та реч није тешка, вршиће се даље проналажење лексикографски тешких речи. Пошто су све речи које се добијају проширивањем лаке речи такође лаке, наредне кандидате ћемо одређивати изменом карактера текуће речи. Лексикографски наредну реч добијамо тако што са краја текуће речи уклањамо сва слова која су једнака максималном и затим увећавамо последње слово у речи које није максимално (ако такво слово не постоји, реч није могуће даље лексикографски увећати тако да остане тешка).

Пошто је свака реч за коју треба проверити да ли је тешка добијена изменом последњег карактера неке тешке речи, поново је довољно проверити само да ли та реч има суфикс који се узастопно понавља.

```cpp
#include <iostream>
#include <string>

using namespace std;

// provera da li se sufiks duzine d dva puta uzastopno ponavlja
bool ponovljenSufiks(const string& s, int d) {
  int n = s.length();
  for (int i = 1; i <= d; i++)
    if (s[n-i] != s[n-d-i])
      return false;
  return true;
}

// provera da li postoji neki pravi sufiks koji se dva puta uzastopno
// ponavlja
bool sufiksJeLak(const string& s) {
  int n = s.length();
  for (int d = 1; d <= n/2; d++)
    if (ponovljenSufiks(s, d))
      return true;
  return false;
}

// za datu rec s pronalazi sledecu tesku rec i vraca informaciju o
// tome da li je uspelo
bool sledecaTeskaRec(string &s, int brSlova) {
  // prvi kandidat za sledecu tesku rec
  s += 'A';
  int n = s.length();
  // dok god rec nije teska i dok postoji sledeca
  while (n > 0 && sufiksJeLak(s)) {
    // prelazimo na leksikografski sledecu rec koja nije prosirenje tekuce
    while (n > 0 && s[n-1] == 'A' + brSlova - 1) {
      s.pop_back();
      n--;
    }
    if (n > 0)
      s[n-1]++;
  }
  // vracamo false ako ne postoji sledeca teska rec
  return n != 0;
}

int main() {
  int brojSlova, k;
  cin >> brojSlova >> k;

  // krecemo od nulte reci i k puta prelazimo na sledecu tesku rec
  // (ako je to moguce)
  string s = "";
  while (k > 0 && sledecaTeskaRec(s, brojSlova))
    k--;

  // ako smo k puta uspeli da nadjemo tesku rec, rezultat je s
  if (k > 0)
    cout << "ne postoji" << endl;
  else
```

```
    cout << s << endl;

  return 0;
}
```

Задатак: Сви распореди заграда

Написати програм којим се за дато n приказују сви исправни распореди n парова заграда у обрнутом лексикографском поретку. На пример за $n = 3$ тражени рапореди су `()()()`, `()(())`, `(())()`, `(()())` и `((()))`.

Улаз: Прва и једина линија стандардног улаза садржи природан број $n \leq 13$.

Излаз: На стандардном излазу приказати распореде n парова заграда у обрнутом лексикографском пореттку, сваки распоред у посебној линији.

Пример

Улаз	*Излаз*

4	`()()()()`
	`()()(())`
	`()(())()`
	`()(()())`
	`()((()))`
	`(())()()`
	`(())(())`
	`(()())()`
	`(()()())`
	`(()(()))`
	`((()))()`
	`((())())`
	`(((())))`

Решење

Један начин да се задатак реши је да се примени рекурзија. Прошириваћемо текућу ниску која садржи заграде заградама, све док је не попунимо са n парова заграда (тј. док јој дужина не постане $2n$). На текуће место i разматрамо могућност да се постави прво затворена, а затим и отворена заграда. Затворену заграду можемо поставити ако у раније попуњеном делу ниске имамо бар једну отворену заграду која још није затворена, а отворену заграду можемо ставити ако у досадашњем делу ниске није укупно отворено n заграда. Након постављања неке заграде на позицију i рекурзивно позивамо функцију да постави заграду на место $i + 1$. Стога је пожељно као параметре рекурзивне функције прослеђивати и укупан број до сада отворених заграда и укупан број до сада затворених заграда (на почетку су оба броја 0). Рекурзија се успешно завршава када су све позиције попуњене и тада се текућа ниска исписује на стандардни излаз.

```
#include <iostream>
#include <string>

using namespace std;

void rasporediZagrade(string& z, int n, int brOtvorene, int brZatvorene) {
  if (brZatvorene == n) {
    cout << z << endl;
    return;
  }
  if (brOtvorene > brZatvorene) {
    z.push_back(')');
    rasporediZagrade(z, n, brOtvorene, brZatvorene+1);
    z.pop_back();
  }
  if (brOtvorene < n) {
    z.push_back('(');
```

```
    rasporediZagrade(z, n, brOtvorene+1, brZatvorene);
    z.pop_back();
  }
}

void rasporediZagrade(int n) {
  string z = "";
  z.reserve(2*n);
  rasporediZagrade(z, n, 0, 0);
}

int main() {
   int n;
   cin >> n;
   rasporediZagrade(n);
   return 0;
}
```

Задатак је могуће решити и без рекурзије, дефинисањем функције која на основу датог одређује наредни распоред заграда у траженом лексикографском редоследу, слично као у задатку Следећа варијација. Циљ нам је да неку што каснију затворену заграду претворимо у отворену. Крећемо од краја и анализирамо затворене заграде. Последњу заграду (која је сигурно затворена) не можемо променити у отворену (јер не бисмо имали када да је затворимо). Слично важи и за све затворене заграде иза којих се налази друга затворена заграда. Мењаћемо дакле прву здесна затворену заграду иза које следи отворена заграда. Ако таква не постоји, тада смо сигурни да смо достигли `((..()..))`, и да наредни редослед не постоји. Када затворену заграду иза које је следила отворена претворимо у отворену, попуњавамо заграде до краја ниске. Да би редослед био што мањи лексикографски, форсираћемо затворене заграде, када год је то могуће. За то је потребно да знамо да ли и колико отворених а незатворених заграда претходи текућој позицији. Ако је тај број позитиван (ако постоји нека незатворена заграда), тада стављамо затворену заграду (и смањујемо тај бројач), а у супротном морамо да ставимо нову отворену заграду (и да повећамо бројач).

```
#include <iostream>

using namespace std;

bool sledeciRaspored(string& s, int n) {
  // pronalazimo poslednju poziciju gde se javlja )(
  int i = 2 * n - 1;
  while (i > 0 && !(s[i-1] == ')' && s[i] == '('))
    i--;
  // ako takva ne postoji, tada je dostignut poslednji raspored ((...(()).....))
  if (i == 0) return false;
  // menjamo zatvorenu zagradu otvorenom
  s[i - 1] = '(';
  // prebrojavamo otvorene, a nezatvorene zagrade zakljucno sa ovom novom otvorenom
  int brojOtvorenih = 0;
  for (int j = 0; j < i; j++)
    if (s[j] == '(')
      brojOtvorenih++;
    else
      brojOtvorenih--;
  // popunjavamo nisku do kraja a u suprotnom
  for (; i < 2*n; i++) {
    if (brojOtvorenih > 0) {
      // kad god postoji otvorena zagrada koja nije zatvorena zatvaramo je
      s[i] = ')';
      brojOtvorenih--;
    } else {
      // u suprotnom smo primorani da stavimo otvorenu zagradu
```

```
      s[i] = '(';
      brojOtvorenih++;
    }
  }
  // uspeli smo da pronadjemo novi redosled
  return true;
}

void rasporediZagrade(int n)
{
  // krecemo od rasporeda ()()...()()
  string s;
  for (int i = 0; i < n; i++)
    s += "()";
  // dok god postoji sledeci raspored, stampamo ga
  do {
    cout << s << endl;
  } while(sledeciRaspored(s, n));
}

int main() {
  int n;
  cin >> n;
  rasporediZagrade(n);
  return 0;
}
```

Још један начин да се генеришу сви распореди заграда је да се примети да је прва заграда увек отворена, тј. да сваки распоред почиње једним паром заграда унутар којега је угнежђен неки распоред заграда у коме учествује између 0 и $n-1$ парова заграда, док иза тог првог пара заграда долази редослед у којима учествују преостали парови заграда (ако је унутар k парова, тада је иза $n-k-1$ парова заграда). Нажалост, овако се не добија лексикографски сортиран редослед. Такође, имплементација мора бити мало компликованија, јер све распореде заграда смештамо у меморију.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

vector<string> rasporediZagrade(int n) {
  vector<string> rezultat;
  if (n == 0) {
    rezultat.push_back("");
  } else {
    for (int i = 0; i < n; i++) {
      vector<string> unutra = rasporediZagrade(i);
      vector<string> spolja = rasporediZagrade(n-i-1);
      for (auto u: unutra)
        for (auto s: spolja)
          rezultat.push_back("(" + u + ")" + s);
    }
  }
  return rezultat;
}

int main() {
  int n;
  cin >> n;
```

```
  for (auto z: rasporediZagrade(n))
    cout << z << endl;
  return 0;
}
```

Задатак: Варијације без понављања

Варијација класе k без понављања елемената скупа S је сваки уређена k - торка од k различитих елемената скупа S. Написати програм који за дато n и k приказује све варијације без понављања класе k скупа бројева од 1 до n, у лексикографском поретку.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 8$), у другој линији налази се природан број k ($0 < k \leq n$).

Излаз: На стандрадном излазу приказати у лексикографском поретку све варијације класе k бројева од 1 до n (сваку у посебном реду).

Пример

Улаз	*Излаз*
3	1 2
2	1 3
	2 1
	2 3
	3 1
	3 2

Решење

Један начин да се задатак реши је да се дефинише функција која за дату варијацију проналази следећу варијацију у лексикографском поретку. Алгоритам представља модификацију алгоритма описаном у задатку Следећа варијација. Полазимо од краја варијације тражећи позицију на којој се налази неки елемент који се може увећати. Да би увећање елемента a_i било могуће, мора постојати неки елемент који је строго већи од a_i (а мањи или једнак n), који се не јавља пре позиције i (јер дупликати нису дозвољени). Ако таква позиција не постоји, тада је варијација лексикографски највећа. У супротном увећавамо елемент на позицију a_i (на најмању могућу вредност), а елементе иза њега попуњавамо редом што мањим елементима скупа $\{1, \ldots, n\}$, који се нису појављивали у дотадашњем делу низа. Да бисмо ефикасније одређивали елементе који су већ употребљени у првом делу варијације, посебно ћемо одржавати скуп тих елемената (најбоље у облику низа логичких вредности тако да вредност на месту i говори да ли је елемент i већ употребљен).

На пример, за $n = 5$ и $k = 5$, следећа варијација у односу на $1, 4, 3, 5, 2$ је $1, 4, 5, 2, 3$. Наиме, елемент 2 не може да се увећа (јер су већи елементи 3, 4 и 5 сви већ употребљени испред њега), елемент 5 не може да се увећа (јер од њега нема већих елемената), док елемент 3 може да се увећа на 5 (не може на 4, јер је елемент 4 већ употребљен испред њега). Након увећања иза се слажу редом елементи 2 и 3 (јер је 1 већ употребљен).

```
#include <iostream>
#include <vector>

using namespace std;

// prikaz date varijacije na standardni izlaz
void prikazi(const vector<int>& a) {
  for (int x : a)
    cout << x << " ";
  cout << endl;
}

// pronalazi i vraca prvi element u intervalu (ai, n] koji nije upotrebljen
// tj. -1 ako takav element ne postoji
int veciNeupotrebljen(int ai, int n, const vector<bool>& upotrebljen) {
  for (int x = ai+1; x <= n; x++)
    if (!upotrebljen[x])
```

```
      return x;
  return -1;
}

bool sledecaVarijacija(vector<int>& a, vector<bool>& upotrebljen, int n) {
  // broj elemenata varijacije
  int k = a.size();

  // analiziramo jednu po jednu poziciju od kraja i pokusavamo da
  // pronadjemo poziciju i takvu da joj se vrednost moze uvecati
  // tj. takvu da postoji element ai_novo > a[i] koji se ne javlja
  // nigde ispred pozicije i
  int i, ai_novo;
  for (i = k-1; i >= 0; i--) {
    ai_novo = veciNeupotrebljen(a[i], n, upotrebljen);
    if (ai_novo == -1)
      upotrebljen[a[i]] = false;
    else
      break;
  }
  // ako se ni jedan element ne moze uvecati, nasli smo leksikografski
  // najvecu varijaciju
  if (i < 0) return false;

  // menjamo element a[i] vrednoscu ai_novo
  upotrebljen[a[i]] = false;
  upotrebljen[ai_novo] = true;
  a[i++] = ai_novo;

  // elemente do kraja varijacije popunjavamo sto manjim,
  // neupotrebljenim elementima
  for (int x = 1; i < k; x++)
    if (!upotrebljen[x]) {
      a[i++] = x;
      upotrebljen[x] = true;
    }

  // uspeli smo da konstruisemo leksikografski narednu varijaciju
  return true;
}

// ispisuje sve varijacije bez ponavljanja k elemenata skupa {1, ..., n}
void varijacije(int n, int k) {
  // elementi varijacije
  vector<int> a(k);
  // za svaki element od 1 do n belezimo da li je upotrebljen u
  // tekucoj varijaciji
  vector<bool> upotrebljen(n+1, false);

  // krecemo od leksikografski najmanje varijacije 1, 2, ..., k
  for (int i = 0; i < k; i++) {
    a[i] = i+1;
    upotrebljen[i+1] = true;
  }

  // ispisujemo tekucu varijaciju i prelazimo na sledecu, sve dok ne
  // dodjemo do leksikografski najvece (koja nema sledecu)
  do {
```

```
    prikazi(a);
  } while (sledecaVarijacija(a, upotrebljen, n));
}

int main() {
  int n, k;
  cin >> n >> k;
  varijacije(n, k);
  return 0;
}
```

Можемо дефинисати и рекурзивну функцију која набраја све варијације без понављања (слично као у задатку Све варијације). Функција прима до сада попуњени део варијације и покушава да постави елемент на позицију i. Ако је $i = k$, тада је цела попуњена и исписујемо је. У супротном на место i редом постављамо један по један елемент скупа $\{1, \ldots, n\}$ који није раније употребљен у варијацији и рекурзивно прелазимо на попуњавање варијације од позиције $i + 1$.

```
#include <iostream>
#include <vector>

using namespace std;

// prikaz date varijacije na standardni izlaz
void prikazi(const vector<int>& a) {
  for (int x : a)
    cout << x << " ";
  cout << endl;
}

// popunjava se varijacija a od pozicije i nadalje elementima skupa
// {1, ..., n} pri cemu se u nizu upotrebljen beleze upotrebljeni
// elementi u delu varijacije pre pozicije i
void varijacije(vector<int>& a, int n, vector<bool>& upotrebljen, int i) {
  // varijacija je cela popunjena, pa je ispisujemo
  if (i == a.size())
    prikazi(a);
  else {
    // na poziciju i stavljamo redom svaki neupotrebljen element
    for (int x = 1; x <= n; x++)
      if(!upotrebljen[x]) {
        a[i] = x;
        upotrebljen[x] = true;
        varijacije(a, n, upotrebljen, i+1);
        upotrebljen[x] = false;
      }
  }
}

// ispisuje sve varijacije k elemenata skupa {1, ..., n}
void varijacije(int n, int k) {
  vector<int> a(k);
  vector<bool> upotrebljen(n+1, false);
  varijacije(a, n, upotrebljen, 0);
}

int main() {
  int n, k;
  cin >> n >> k;
  varijacije(n, k);
```

```
  return 0;
}
```

Задатак: Разлика суседних цифара k

Написати програм којим се за дато n и k, приказују сви природни n-тоцифрени бројеви такви да им је разлика две суседне цифре једнака датом броју k ($0 \leq k \leq 4$). На пример у броју 5753 разлика сваке две суседне цифре једнака је 2.

Улаз: Прва линија стандардног улаза садржи природан број n $0 < n < 10$, друга линија садржи природан број k ($0 \leq k \leq 9$).

Излаз: Приказати тражене бројеве у растућем поретку, сваки број у посебној линија.

Пример

Улаз	*Излаз*
3	131
2	135
	202
	242
	246
	313
	353
	357
	420
	424
	464
	468
	531
	535
	575
	579
	642
	646
	686
	753
	757
	797
	864
	868
	975
	979

Решење

Задатак је могуће решити дефинисањем рекурзивне функције која генерише све тражене бројеве. Функција прима низ цифара, чијих је првих i елемената већ попуњено и попуњава остатак низа. Ако је i једнако дужини низа, цео низ је попуњен, па се тренутни низ само исписује. У супротном се анализирају могућности за цифру на позицији i. Претпосавићемо да је $i > 0$, тј. да знамо вредност претходне цифре c. Тада се на текућу позицију може ставити или вредност $c-k$ (ако је већа или једнака од 0) или вредност $c+k$ (ако је мања или једнака од 9). У случају када је $k = 0$, цифре $c - k$ и $c + k$ се поклапају (обе су једнаке c) па је довољно анализирати само једну од њих. Након постављања цифре на позицију i, рекурзивно попуњавамо остатак низа (прослеђујући вредност $i + 1$). Прва цифра је специфична и њу ћемо попунити пре првог рекурзивног позива. На прво место можемо ставити било коју цифру осим нуле и за сваки избор те прве цифре вршимо посебан рекурзивни позив за $i = 1$.

```
#include <iostream>

using namespace std;

void prikazi(int a[], int n) {
```

```
  for (int i = 0; i < n; i++)
    cout << a[i];
  cout << endl;
}

void brojeviSaRazlikomSusednihCifaraK(int a[], int n, int k, int i) {
  if (i == n) {
    prikazi(a, n);
    return;
  }

  if (a[i - 1] - k >= 0) {
    a[i] = a[i - 1] - k;
    brojeviSaRazlikomSusednihCifaraK(a, n, k, i+1);
  }

  if (k > 0 && a[i - 1] + k < 10) {
    a[i] = a[i-1] + k;
    brojeviSaRazlikomSusednihCifaraK(a, n, k, i+1);
  }
}

void brojeviSaRazlikomSusednihCifaraK(int n, int k) {
  int a[10];
  for(int i = 1; i < 10; i++) {
    a[0] = i;
    brojeviSaRazlikomSusednihCifaraK(a, n, k, 1);
  }
}

int main() {
  int n, k;
  cin >> n >> k;
  brojeviSaRazlikomSusednihCifaraK(n, k);
  return 0;
}
```

Задатак: Комплетан Грејов код

Грејов кôд реда n подразумева ређање свих n-тоцифрених бинарних записа тако да се свака два суседна записа разликују тачно у једном биту (при чему ово важи и за први и последњи запис, тако да се може сматрати да су сви записи поређани у круг).

Грејов кôд дужине 0 садржи само један елемент и то празну ниску. Грејов кôд дужине $n+1$ се може добити од кода дужине n тако што се испред сваког броја у коду дужине n допише цифра 0, затим се редослед елемената у коду дужине n обрне и на сваком броју се на почетак допише цифра 1 и два тако добијена низа бројева се споје. Нпр. Грејов кôд реда 2 је

```
00
01
11
10
```

На основу претходног поступка добијамо Грејов кôд реда 3.

```
0 00  |                k
0 01  |                0:  000
0 11  |                1:  001
0 10  v                2:  011
            tj.        3:  010
```

```
1 10  ^                4:  110
1 11  |                5:  111
1 01  |                6:  101
1 00  |                7:  100
```

Напиши програм који за дату дужину кода n исписује комплетан Грејов код дужине n.

Улаз: Са стандардног улаза се учитава дужина кода n ($1 \leq n \leq 14$).

Излаз: На стандардни излаз исписати тражени код.

Пример 1

Улаз	*Излаз*
3	000
	001
	011
	010
	110
	111
	101
	100

Пример 2

Улаз	*Излаз*
4	0000
	0001
	0011
	0010
	0110
	0111
	0101
	0100
	1100
	1101
	1111
	1110
	1010
	1011
	1001
	1000

Решење

На основу дефиниције Грејовог кода закључујемо да је погодно дефинисати рекурзивну функцију за исписивање кода мало општије. Наиме, омогућићемо да функција испред сваке линије кода испише задати префикс, као и да линије кода испише у директном или обрнутом редоследу.

Према томе, функција ће имати три параметра: ниску `prefiks`, целобројну дужину кода `n` и логичку вредност `unapred`, која задаје редослед исписивања линија кода (у директном редоследу за `unapred = true`, а у обрнутом иначе).

Код дужине 0 се састоји само од једне празне ниске, па у случају да је дужина кода `n = 0`, функција исписује само префикс.

За `n > 0`, разликујемо два подслучаја.

- Ако код треба исписати у директном редоследу (`unapred = true`), исписујемо директно линије кода дужине `n-1` са префиксом проширеним цифром 0, а затим обрнуто линије кода дужине `n-1` са префиксом проширеним цифром 1.
- У случају да код треба исписати у обрнутом редоследу (`unapred = false`), исписујемо директно (обрнуто од обрнутог) линије кода дужине `n-1` са префиксом проширеним цифром 1, а затим обрнуто линије кода дужине `n-1` са префиксом проширеним цифром 0.

```
#include <iostream>
#include <string>

using namespace std;

void grej(string prefiks, int n, bool unapred) {
    if (n == 0)
        cout << prefiks << endl;
    else if (unapred)
    {
        grej(prefiks + '0', n-1, true);
        grej(prefiks + '1', n-1, false);
```

```
    }
    else
    {
        grej(prefiks + '1', n-1, true);
        grej(prefiks + '0', n-1, false);
    }
}

int main() {
  unsigned n;
  cin >> n;
  grej("", n, true);
  return 0;
}
```

Функцију можемо реализовати и итеративно, и то на више начина. Један начин је да сваку линију кода израчунамо на основу њеног редног броја, као у задатку Грејов код.

Други начин је да искористимо чињеницу да се две узастопне линије кода разликују на само једној позицији (у једном биту). Посматрајући један конкретан код примећујемо да се бит који се мења редом налази на позицијама 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4 итд. бројећи позиције од нуле с десна на лево. Исте ове позиције су уједно и позиције последње јединице у бинарном запису редног броја текуће итерације. На пример, у итерацијама 1, 2, 3, 4, последња јединица је редом на позицијама 0, 1, 0, 2.

Ово значи да је у свакој итерацији потребно издвојити позицију последње јединице у редном броју итерације и на тој позицији променити бит у запису текуће линије кода. Променљива `maska` посматрана као низ битова, садржи 1 само на поменутој позицији бита који треба променити.

```
#include <iostream>
#include <string>

using namespace std;

void print(int x, int n) {
    string rez(n, '0');
    for (int i = n - 1; i >= 0; --i) {
        rez[i] = '0' + (x & 1);
        x /= 2;
    }
    cout << rez << endl;
}

void grej(int n) {
    int red = 0;
    for(int i = 0; i < (1 << n); ++i)
    {
        int maska = i - (i & (i-1));
        red = red ^ maska;
        print(red, n);
    }
}

int main() {
  unsigned n;
  cin >> n;
  grej(n);
  return 0;
}
```

Глава 5

Бектрекинг и груба сила

Задатак: Пут кроз лавиринт

Напиши програм који испитује да ли се у правоуаоном лавиринту може стићи од горњег левог до доњег десног угла.

Улаз: Са стандардног улаза се учитавају димензије лавиринта m и n раздвојене једним размаком. Након тога се учитава матрица којом је представљен лавиринт. Поља кроз која се може проћи су обележена карактером `.`, а поља на којима је препрека карактером `x`.

Излаз: На стандардни излаз исписати `da` ако пут постоји тј. `ne` ако пут не постоји.

Пример 1

Улаз	*Излаз*

`8 8`
`.x.....x`
`.x.x.x.x`
`.x.x.x.x`
`.x.x.x.x`
`.x.x.x.x`
`.x.x.x.x`
`.x.x.x.x`
`...x.x..`
	`da`

Пример 2

Улаз	*Излаз*

`8 8`
`.x..x..x`
`.x.x.x.x`
`.x.x.x.x`
`.x.x.x.x`
`.x...x.x`
`.x.x.x.x`
`.x.x.x.x`
`...x.x..`
	`ne`

Решење

Претрага у дубину

Задатак решавамо исцрпном претрагом свих могућих путања. Претрагу можемо организовати "у дубину" помоћу рекрузивне функције. Функција прима матрицу препрека и поље на ком се тренутно налазимо, које се мења током рекурзије. Ако је стартно поље поклапа са циљним (доњим десним углом), пут је успешно пронађен. У супротном, испитујемо 4 суседа стартног поља (осим у случају поља на рубу лавиринта, када је суседа мање) и претрагу рекурзивно настављамо од сваког од њих (суседно поље постаје ново стартно поље). Ако се на пољу на које смо прешли налази препрека, претрагу моментално прекидамо јер тај корак није дозвољен (функција враћа да на тај начин није могуће пронаћи пут). Потребно је и да обезбедимо да се већ посећена стартна поља не обрађују поново и за то користимо помоћну матрицу у којој за свако поље региструјемо да ли је посећено или није. Ако приликом позива функције установимо да је поље на које смо дошли већ раније посећено, претрагу одмах прекидамо, док у супротном означавамо да је то поље посећено и прелазимо на анализу њега и његових суседа..

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;
```

```
bool postojiPut(const vector<vector<bool>>& prepreke, int m, int n,
                vector<vector<bool>>& poseceno,
                int v, int k) {

  if (prepreke[v][k] || poseceno[v][k])
    return false;

  poseceno[v][k] = true;

  if (v == m - 1 && k == n - 1)
    return true;

  if (v > 0 && postojiPut(prepreke, m, n, poseceno, v-1, k))
    return true;
  if (v < m-1 && postojiPut(prepreke, m, n, poseceno, v+1, k))
    return true;
  if (k > 0 && postojiPut(prepreke, m, n, poseceno, v, k-1))
    return true;
  if (k < n-1 && postojiPut(prepreke, m, n, poseceno, v, k+1))
    return true;
  return false;
}

bool postojiPut(const vector<vector<bool>>& prepreke, int m, int n) {
  vector<vector<bool>> poseceno(m, vector<bool>(n, false));
  return postojiPut(prepreke, m, n, poseceno, 0, 0);
}

int main() {
  int m, n;
  cin >> m >> n >> ws;
  vector<vector<bool>> prepreke(m);
  for (int i = 0; i < m; i++) {
    prepreke[i].resize(n);
    string red;
    getline(cin, red);
    for (int j = 0; j < n; j++)
      prepreke[i][j] = red[j] == 'x';
  }

  if (postojiPut(prepreke, m, n))
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Претрагу у дубину је могуће имплементирати и нерекурзивно, тако што експлицитно одржавамо стек.

Постоји још неколико једноставних трикова који могу олакшати имплементацију. Четири суседна поља можемо обрађивати у петљи (тако што у низу сачувамо 4 помераја $(-1, 0)$, $(1, 0)$, $(0, -1)$ и $(0, 1)$, чиме избегавамо понављање сличног кода 4 пута. Да би се приликом обиласка 4 суседа избегло испитивање да ли ти суседи постоје тј. да ли је текуће поље на рубу, можемо матрицу проширити оквиром који садржи препреке.

```
#include <iostream>
#include <string>
#include <vector>
#include <stack>
#include <utility>
```

```
using namespace std;

template <typename T>
using Matrica = vector<vector<T>>;

template <typename T>
Matrica<T> napraviMatricu(int m, int n, T vrednost) {
  Matrica<T> matrica(m);
  for (int i = 0; i < m; i++)
    matrica[i].resize(n, vrednost);
  return matrica;
}

bool postojiPut(const Matrica<bool>& prepreke, int m, int n) {
  Matrica<bool> poseceno = napraviMatricu(m + 2, n + 2, false);
  poseceno[1][1] = true;

  stack<pair<int, int>> stek;
  stek.push(make_pair(1, 1));

  while (!stek.empty()) {
    int v = stek.top().first, k = stek.top().second;
    stek.pop();

    if (v == m && k == n)
      return true;

    int pravac[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    for (int i = 0; i < 4; i++) {
      int v1 = v + pravac[i][0], k1 = k + pravac[i][1];
      if (!prepreke[v1][k1] && !poseceno[v1][k1]) {
        poseceno[v1][k1] = true;
        stek.push(make_pair(v1, k1));
      }
    }
  }
  return false;
}

int main() {
  int m, n;
  cin >> m >> n >> ws;
  Matrica<bool> prepreke = napraviMatricu(m + 2, n + 2, true);
  for (int i = 0; i < m; i++) {
    string red;
    getline(cin, red);
    for (int j = 0; j < n; j++)
      prepreke[i+1][j+1] = red[j] == 'x';
  }

  if (postojiPut(prepreke, m, n))
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Претрага у ширину

Уместо претраге у дубину, могуће је употребити и претрагу у ширину. Имплементација је веома слична нерекурзивно имплементираној претрази у дубину, при чему се уместо стека користи ред, чиме се постиже да се поља обрађују у редоследу њиховог растојања од полазног поља, што омогућава да се лако очита и дужина најкраћег пута до крајњег поља.

```
#include <iostream>
#include <string>
#include <vector>
#include <utility>
#include <queue>

using namespace std;

template <typename T>
using Matrica = vector<vector<T>>;

template <typename T>
Matrica<T> napraviMatricu(int m, int n, T vrednost) {
  Matrica<T> matrica(m);
  for (int i = 0; i < m; i++)
    matrica[i].resize(n, vrednost);
  return matrica;
}

bool postojiPut(const Matrica<bool>& prepreke, int m, int n) {
  Matrica<bool> poseceno = napraviMatricu(m, n, false);
  queue<pair<int, int>> red;
  red.push(make_pair(0, 0));
  while (!red.empty()) {
    int v = red.front().first, k = red.front().second;
    red.pop();
    if (poseceno[v][k] || prepreke[v][k])
      continue;
    poseceno[v][k] = true;
    if (v == m-1 && k == n-1)
      return true;
    int pravac[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    for (int i = 0; i < 4; i++) {
      int v1 = v + pravac[i][0], k1 = k + pravac[i][1];
      if (0 <= v1 && v1 < m && 0 <= k1 && k1 < n)
        red.push(make_pair(v1, k1));
    }
  }
  return false;
}

int main() {
  int m, n;
  cin >> m >> n >> ws;
  Matrica<bool> prepreke = napraviMatricu(m, n, false);
  for (int i = 0; i < m; i++) {
    string red;
    getline(cin, red);
    for (int j = 0; j < n; j++)
      prepreke[i][j] = red[j] == 'x';
  }
```

```
  if (postojiPut(prepreke, m, n))
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Задатак: Максимални безбедни пут

Дата је матрица A_{nxm} у којој се налазе мине означене са 0, остала поља садрже природне бројеве. Треба из било ког поља у првој колони доћи на било које поље у последњој колони матрице безбедним путем. Пут је безбедан ако не садржи мину и не садржи поље коме је сусед, горњи, доњи, леви или десни, мина. Кроз матрицу можемо да се крећемо у четри правца горе, доле, десно и лево и при томе на једно поље можемо стати највише једанпут. Одреди максимални збир бројева који се могу наћи на неком безбедном путу од прве до последње колоне.

Улаз: Прва линија стандардног улаза садржи димензије матрице, два природна броја, m и n ($1 < m, n \leq 20$). На стандардном улазу у следећих n линија налазе се по m целих бројева већих или једнаких 0, бројеви су одвојени са по једним бланко карактером, и престављају редом елементе матрице.

Излаз: На стандардном излазу приказати природан број који представља максималну суму безбедног пута од прве до последње колоне. Ако безбедан пут не постоји приказати -1.

Пример

Улаз	*Излаз*
4 4 5 1 2 0 1 2 3 4 2 4 1 1 0 1 1 1	17

Објашњење

$$1 + 5 + 1 + 2 + 4 + 1 + 1 + 1 + 1 = 17$$

Решење

У фази претпроцесирања у матрици ћемо обележити сва небезбедна поља (то можемо урадити тако што ћемо анализирати сваки елемент матрице и за сваку мину коју нађемо, обележићемо њена 4 суседна поља на којима није мина).

Дефинисаћемо рекурзивну функцију која одређује максимални збир бројева на путањи од датог текућег поља до неког поља у последњом врсти матрице. Максимум иницијализујемо на -1 (што значи да путања до последње врсте још није пронађена). Ако се текуће поље налази у последњој врсти, тада ажурирамо максимум на вредност текућег поља (али не враћамо коначан резултат, јер је могуће да се још померамо и дођемо на неко друго поље на доњој ивици). Након тога анализирамо четири суседна поља текућем. Рекурзивно израчунавамо резултат за свако од њих и максимум ажурирамо ако пронађемо да је са неког од њих било могуће стићи до дна преко неке путање која има већи збир од текућег максимума. При том морамо да водимо рачуна да се не враћамо на поља која смо посетили (што можемо постићи тако што памтимо већ посећена поља било у посебној матрици, било модификујући вредности у полазној матрици).

Напоменимо да је због мноштва потенцијалних путања описани рекурзивни алгоритам веома неефикасан.

```
#include <iostream>
#include <vector>

using namespace std;

// matrica i njene dimenzije
int bv, bk;
vector<vector<int>> A;
```

```
// pomeraji koji odredjuju 4 susedna polja
int dv[] = {1, -1, 0, 0};
int dk[] = {0, 0, 1, -1};

const int MINA = 0;
const int NEBEZBEDNO = -1;
const int POSECENO = -2;

// ucitavanje matrice sa standardnog ulaza
void citajMatricu() {
  cin >> bv >> bk;
  A.resize(bv, vector<int>(bk));
  for (int v = 0; v < bv; v++)
    for (int k = 0; k < bk; k++)
      cin >> A[v][k];
}

// provera da li se polje (v, k) nalazi unutar matrice
bool uMatrici(int v, int k) {
  return 0 <= v && v < bv &&
         0 <= k && k < bk;
}

// u matrici se obelezavaju sva nebezbedna polja (ona ciji su susedi mine)
void oznaciNebezbednaPolja() {
  for (int v = 0; v < bv; v++)
    for (int k = 0; k < bk; k++)
      if (A[v][k] == MINA)
        for (int i = 0; i < 4; i++) {
          int vv = v + dv[i], kk = k + dk[i];
          if (uMatrici(vv, kk) && A[vv][kk] != MINA)
            A[vv][kk] = NEBEZBEDNO;
        }
}

// put sa najvecim zbirom od polja (v, k) do donjeg reda matrice
int maksPut(int v, int k) {
  int maks = -1;
  if (v == bv-1)
    maks = A[v][k];

  int tmp = A[v][k];
  A[v][k] = POSECENO;
  for (int i = 0; i < 4; i++) {
    int vv = v + dv[i], kk = k + dk[i];
    if (uMatrici(vv, kk) && A[vv][kk] > 0) {
      int put = maksPut(vv, kk);
      if (put != -1)
        maks = max(maks, put + tmp);
    }
  }
  A[v][k] = tmp;
  return maks;
}

int main() {
  citajMatricu();
```

```
  oznaciNebezbednaPolja();
  // najveci zbir koji se moze prikupiti
  int maks = -1;
  // posebno pokrecemo pretragu iz svakog polja u prvom redu matrice
  for (int k = 0; k < bk; k++) {
    if (A[0][k] > 0) {
      int put = maksPut(0, k);
      if (put != -1)
        maks = max(maks, put);
    }
  }
  // ispisujemo najpovoljniji put
  cout << maks << endl;

  return 0;
}
```

Задатак: Обојени лавиринт

Испред тебе се налази необична стаза. Стаза се састоји од поља задатих матрицом и свако поље је је обојено једном од 4 боје (црвена поља су обележена бројем 1, плава бројем 2, жута бројем 3 и зелена бројем 4). Правила кретања су таква да се мора почи са црвеног поља, са црвеног поља се може прећи само на плаво, са плавог на жуто, са жутог на зелено и са зеленог на црвено (мора се почи са броја 1 и током кретања бројеви морају бити 1, 2, 3, 4, 1, 2, 3, 4, …). Напиши програм који одређује да ли се од доњег реда стазе може стићи до горњег.

Улаз: Са стандардног улаза се уносе димензије стазе m и n ($1 \leq m, n \leq 20$), раздвојене размаком. Након тога се уноси матрица попуњена бројевима од 1 до 4 (без размака између њих).

Излаз: На стандардни излаз исписати `da`, ако пут постоји тј. `ne`, ако пут не постоји.

Пример

Улаз	*Излаз*

`5 6`
`143222`
`414344`
`323244`
`124123`
`343211`
	`da`

Објашњење

Један од путева којим се може стићи је следећи.

```
5 6
1.....
4143..
32.2..
..41..
..321.
```

Решење

Задатак решавамо рекурзивном претрагом у дубину. Рекурзивна функција прима поље на којем се тренутно налазимо. Ако се оно налази у највишој врсти, пут је успешно пронађен. У супротном, анализирамо суседна поља која раније нисмо посетили (посете региструјемо посебном матрицом) а на којима се налази наредни број и померамо се на свако од њих (чим се за неко одреди да успешно води до врха, можемо прекинути даље испитивање). Ако бројеве приликом учитавања пребацимо на бројеве од 0 до 3, онда проверу да ли је следећи број одговарајући можемо извршити тако што проверимо да ли се на том следећем пољу налази број који се од тренутног добија увећањем за један и проналажењем остатка при дељењу са 4 (тј. бројем боја).

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

bool postojiPut(const vector<vector<int>>& polja, int m, int n,
                int v, int k, vector<vector<bool>>& poseceno) {
  poseceno[v][k] = true;

  if (v == 0)
    return true;

  int pravac[4][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
  for (int i = 0; i < 4; i++) {
    int v1 = v + pravac[i][0], k1 = k + pravac[i][1];
    if (0 <= v1 && v1 < m && 0 <= k1 && k1 < n &&
        polja[v1][k1] == (polja[v][k] + 1) % 4 &&
        !poseceno[v1][k1] &&
        postojiPut(polja, m, n, v1, k1, poseceno))
      return true;
  }
  return false;
}

bool postojiPut(const vector<vector<int>>& polja, int m, int n) {
  vector<vector<bool>> poseceno(m, vector<bool>(n, false));
  for (int k = 0; k < n; k++)
    if (polja[m-1][k] == 0 && postojiPut(polja, m, n, m-1, k, poseceno))
      return true;
  return false;
}

int main() {
  int m, n;
  cin >> m >> n >> ws;
  vector<vector<int>> polja(m);
  for (int i = 0; i < m; i++) {
    polja[i].resize(n);
    string str;
    getline(cin, str);
    for (int j = 0; j < n; j++)
      polja[i][j] = str[j] - '1';
  }

  if (postojiPut(polja, m, n))
    cout << "da" << endl;
  else
    cout << "ne" << endl;

  return 0;
}
```

Задатак: Падајућа лоптица

Лоптица се испушта са висине и пада кроз простор испуњен препрекама. Када испод ње нема препрека, она пада директно наниже. Када је препрека испод ње, она се зауставља и тада је можемо гурнути било лево, било десно до ивице препреке, након чега она наставља да пада. Напиши програм који одређује да ли лоптица може

да дође до дна.

Улаз: Са стандардног улаза се учитава матрица карактера до краја улаза (сви редови стандардног улаза су исте дужине и представљају редове матрице). Ни дужина ни ширина матрице нису већи од 16. Препреке су означене карактерима `x`, празнине карактерима `.`, док је почетни положај лоптице одређен карактером `O` и налази се у највишој врсти.

Излаз: На стандардни излаз исписати `da` ако лоптица може да падне до дна тј. `ne` ако не може.

Пример 1

Улаз	*Излаз*
`O.......x...x`	`ne`
`xxx.....xx..x`	
`xxx....xxx.xx`	
`xxxx........x`	
`xxxx..xx....x`	
`xxxxxxxx.xxxx`	

Пример 2

Улаз	*Излаз*
`.O......x...x`	`da`
`xxx.....xx..x`	
`.......xxx.xx`	
`.xxx........x`	
`.xxxxx.x....x`	
`.xxxxxxx.xxxx`	

Решење

Решење градимо рекурзивном претрагом у дубину. Рекурзивна функција која испитује да ли лоптица може да падне до дна прима матрицу препрека и тренутни положај лоптице. На почетку функције проверавамо да ли се лоптица налази у последњој врсти и ако се налази, враћамо информацију да је лоптица пала. У супротном, проверавамо да ли се испод лоптице налази препрека. Ако се не налази, тада лоптицу померамо наниже и враћамо резултат који нам врати рекурзивни позив за тако померену лоптицу. Ако се налази, тада вршимо два рекурзивна позива - један у коме лоптицу померамо налево, а други за који лоптицу померамо надесно. Ако нам било који од њих јави да је лоптица пала до дна, враћамо вредност тачно, а у супротном враћамо вредност нетачно. Да бисмо избегли да се лоптица стално помера лево-десно, у помоћној матрици памтимо посећена поља и ако је неко поље већ посећено, прескачемо рекурзивни позив (на уласку у функцију у помоћној матрици записујемо да је тренутно поље посећено).

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

bool mozePasti(const vector<string>& prepreke, int bv, int bk,
               vector<vector<bool>>& poseceno,
               int v, int k) {
  poseceno[v][k] = true;
  if (v == bv - 1)
    return true;
  if (prepreke[v+1][k] != 'x')
    return mozePasti(prepreke, bv, bk, poseceno, v+1, k);
  if (k > 0 && prepreke[v][k-1] != 'x' && !poseceno[v][k-1] &&
      mozePasti(prepreke, bv, bk, poseceno, v, k-1))
    return true;
  if (k < bk - 1 && prepreke[v][k+1] != 'x' && !poseceno[v][k+1] &&
      mozePasti(prepreke, bv, bk, poseceno, v, k+1))
    return true;
  return false;
}

bool mozePasti(const vector<string>& prepreke) {
  int bv = prepreke.size();
  int bk = prepreke[0].size();
  vector<vector<bool>> poseceno(bv, vector<bool>(bk, false));
  return mozePasti(prepreke, bv, bk, poseceno, 0, prepreke[0].find('O'));
}

int main() {
```

```
  ios_base::sync_with_stdio(false);
  vector<string> prepreke;
  string red;
  while (getline(cin, red))
    prepreke.push_back(red);

  if (mozePasti(prepreke))
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Задатак: Распоређивање n дама на шаховској табли

Напиши програм који одређује све положаје n дама на шаховској табли димензије $n \times n$ такве да се никоје две даме међусобно не нападају. Да се даме не би нападале у свакој врсти мора бити тачно једна дама, при чему никоје две даме нису у истој колони. Распоред је зато одређен низом од n различитих бројева од 1 до n који редом представљају бројеве колона у којима се даме налазе у врстама од 1 до n.

Улаз: Са стандардног улаза се уноси број n ($4 \leq n \leq 11$).

Излаз: На стандардни излаз исписати све могуће распореде дама (у произвољном редоследу).

Пример

Улаз	*Излаз*
4	3 1 4 2
	2 4 1 3

Решење

Груба сила - провера свих пермутација

Један (наиван) начин да се одреде сви могући распореди је да се грубом силом наброје сви могући распореди, да се испита који од њих представљају исправан распоред (у ком се даме не нападају) и да се испишу само они који тај критеријум задовољавају. Важно питање је репрезентација позиција. Најбоље решење се добија ако се распореди представе пермутацијама елемената скупа $\{1, 2, \ldots, n\}$. Наиме ако се даме не нападају, у свакој врсти и у свакој колони се налази по тачно једна дама. Ако и врсте и колоне обележимо бројевима од 0 до $n - 1$, тада је свакој колони једнозначно придружена врста у којој се налази дама у тој колони и распоред можемо представити низом тих бројева. Свакој колони је придружена различита врста (јер даме не смеју да се нападају), тако да је заиста у питању пермутација. Овај распоред у старту гарантује да се даме неће нападати ни хоризонтално, ни вертикално и једино је потребно одредити да ли се нападају по дијагонали. Две даме се налазе на истој дијагонали ако и само ако је хоризонтални размак између колона у којима се налазе једнак вертикалном размаку врста (као што је приказано у задатку Да ли се даме нападају). За сваки пар дама проверавамо да ли се нападају дијагонално. Све пермутације можемо набројати на било који од начина описаних у задатку Све пермутације.

```
#include <iostream>
#include <vector>

using namespace std;

bool dameSeNapadaju(const vector<int>& permutacija) {
  for (int i = 0; i < permutacija.size(); i++)
    for (int j = i + 1; j < permutacija.size(); j++)
       if (abs(i - j) == abs(permutacija[i] - permutacija[j]))
          return true;
  return false;
}

void obradi(const vector<int>& permutacija) {
```

```
  if (!dameSeNapadaju(permutacija)) {
     for (int x : permutacija)
        cout << x << " ";
     cout << endl;
  }
}

void obradiSvePermutacije(vector<int>& permutacija, int k) {
  if (k == 1)
    obradi(permutacija);
  else {
    for (int i = 0; i < k; i++) {
      swap(permutacija[i], permutacija[k-1]);
      obradiSvePermutacije(permutacija, k-1);
      swap(permutacija[i], permutacija[k-1]);
    }
  }
}

void obradiSvePermutacije(int n) {
  vector<int> permutacija(n);
  for (int i = 1; i <= n; i++)
    permutacija[i-1] = i;
  obradiSvePermutacije(permutacija, n);
}

int main() {
  int n;
  cin >> n;
  obradiSvePermutacije(n);
  return 0;
}
```

Бектрекинг и одсецање

Прикажимо сада решење проблема постављања n дама на шаховску таблу, ког смо већ решавали грубом силом. Основна разлика овог у односу на претходно решење биће то што се коректност пермутација неће проверавати тек након што су генерисане у целости, већ ће бити проверавана коректност и сваке делимично попуњене пермутације. У многим случајевима веома рано ће бити откривено да су постављене даме на истој дијагонали и цела та грана претраге биће напуштена, што даје потенцијално велике добитке у ефикасности.

Основна рекурзивна функција прима вектор у коме се на позицијама из интервала $[0, k)$ налазе даме које су постављене у првих k колона и задатак функције ће бити да испише све могуће распореде који проширују тај (додајући преостале даме у преостале колоне). Важна инваријанта ће бити да се даме које су постављене у тих првих k колона не нападају. Ако је $k = n$, тада су све даме већ постављене, на основу инваријанте знамо да се не нападају и можемо да испишемо то решење (оно је јединствено и не може се проширити). У супротном, разматрамо све опције за постављање даме на позицију k, тако да се даме у тако проширеном скупу не нападају. Пошто се зна да се даме на позицијама $[0, k)$ не нападају потребно је само проверити да ли се дама на позицији k напада са неком од дама постављених у првих k колона. Размотримо шта су кандидати за вредности на позицији k. Пошто не смемо имати два иста елемента низа тј. две исте врсте на којима се налазе даме, могли бисмо у делу низа на позицијама $[k, n)$ одржавати скуп елемената који су потенцијални кандидати за позицију k. Међутим, пошто за проверу дијагонала морамо упоредити даму k са свим претходно постављеним дамама, имплементацију можемо олакшати тако што на позицију k стављамо шири скуп могућих кандидата (скуп свих врста од 0 до $n - 1$) а онда за сваки од тих бројева проверавамо да ли се јавио у претходном делу низа чиме би се даме нападале по хоризонтали и да ли се постављањем даме на у ту врсту она по дијагонали нападала са неком претходно постављеном дамом. Ако се установи да то није случај, тј. да се постављањем даме у ту врсту она не напада ни са једном од претходно постављених дама, онда је инваријанта задовољена и рекурзивно прелазимо на попуњавање наредних дама. Нема потребе за експлицитним поништавањем одлука које доносемо, јер ће се у свакој новој итерацији допуштена вредност

уписати на позицију k, аутоматски поништавајући вредност коју смо ту раније уписали.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje permutaciju na standardni izlaz
void ispisi(const vector<int>& permutacija) {
  for (int x : permutacija)
    cout << (x+1) << " ";
  cout << endl;
}


// niz permutacija sadrži vrste dama u kolonama iz intervala [0, k]
// pretpostavlja se da se dame na pozicijama [0, k) ne napadaju
// i proverava se da li se dama na poziciji k napada sa nekom od njih
bool dameSeNapadaju(const vector<int>& permutacija, int k) {
  for (int i = 0; i < k; i++) {
    if (permutacija[i] == permutacija[k])
      return true;
    if (abs(k-i) == abs(permutacija[k] - permutacija[i]))
      return true;
  }
  return false;
}

// niz permutacija sadrži vrste dama u kolonama iz intervala [0, k)
// za koji se pretpostavlja da predstavlja raspored dama koje se ne napadaju
// procedura ispisuje sve moguće raspored dama koje proširuju taj raspored
// i u kojima se dame ne napadaju
void nDama(vector<int>& permutacija, int k) {
  // sve dame su postavljene
  if (k == permutacija.size())
    ispisi(permutacija);
  else {
    // postavljamo damu u kolonu k
    // isprobavamo sve moguće vrste
    for (int i = 0; i < permutacija.size(); i++) {
      // postavljamo damu u koloni k u vrstu i
      permutacija[k] = i;
      // proveravamo da li se dame i dalje ne napadaju
      if (!dameSeNapadaju(permutacija, k))
        // ako se ne napadaju, nastavljamo sa proširivanjem tekućeg rasporeda
        nDama(permutacija, k+1);
    }
  }
}

void nDama(int n) {
   // niz koji za svaku kolonu beleži vrstu dame u toj koloni
   vector<int> permutacija(n);
   // krećemo pretragu od prazne table
   nDama(permutacija, 0);
}

int main() {
  ios_base::sync_with_stdio(false); cout.tie(0);
```

```
  int n;
  cin >> n;
  nDama(n);
  return 0;
}
```

Задатак: Судоку

Напиши програм који попуњава Судоку загонетку чији је циљ да се у матрицу димензије 9 пута 9 распореде бројеви од 1 до 9, тако да у свакој врсти, у свакој колони и у сваком од 9 квадрата димензије 3 пута 3 сви бројеви различити.

Улаз: Са стандардног улаза се учитава матрица димензије 9 пута 9 у којој су већ уписани неки бројеви, а на пољима која су празна уписана је нула.

Излаз: На стандардни излаз исписати решење загонетке (тест-примери ће бити такви да је решење сигурно јединствено).

Пример

Улаз	*Излаз*
749030680	749132685
006508000	326548179
000760324	518769324
800057060	892357461
407000508	437621598
050980002	651984732
184076000	184276953
000403800	275493816
063010247	963815247

Решење

Задатак решавамо бектрекингом тј. рекурзивно имплементираном претрагом у дубину (слично као у задатку Латински квадрати). Рекрузивна функција уз матрицу која се попуњава добија и редни број поља које треба попунити (она враћа информацију о томе да ли је матрицу било могуће потпуно попунити). Попуњавање креће од горњег левог угла и тече врсту по врсту, све док не попунимо целу матрицу. Координате поља се веома једноставно могу одредити на основу његовог редног броја (слично као у задатку Шаховско поље). Ако је тренутно поље већ попуњено (јер су у старту нека поља већ попуњена), тада одмах прелазимо на наредно поље. У супротном проверавамо све могуће вредности за то поље. Након уписа вредности проверавамо да ли је тиме направљен неки конфликт тј. да ли се десило да је у истој врсти, у истој колони или у истом квадрату већ постојао број који је уписан. Ако јесте, претрагу прекидамо, а ако није, настављамо је даље, попуњавањем наредног поља. Чим неки од рекурзивних позива успе да попуни целу матрицу, претрага се прекида и наредни рекурзивни позиви се не врше.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

const int n = 3;

bool konflikt(const vector<vector<int>>& A, int i, int j) {
  // da li se A[i][j] nalazi već u koloni j
  for (int k = 0; k < n * n; k++)
    if (k != i && A[i][j] == A[k][j])
      return true;

  // da li se A[i][j] nalazi već u vrsti i
  for (int k = 0; k < n * n; k++)
    if (k != j && A[i][j] == A[i][k])
      return true;
```

```
  // da li se A[i][j] već nalazi u kvadratu koji sadrži polje (i, j)
  int x = i / n, y = j / n;
  for (int k = x * n; k < (x + 1) * n; k++)
    for (int l = y * n; l < (y + 1) * n; l++)
      if ((k != i || l != j) && A[i][j] == A[k][l])
        return true;

  // ne postoji konflikt
  return false;
}

bool sudoku(vector<vector<int>>& A, int rbr) {
  int i = rbr / (n*n), j = rbr % (n*n);
  // ako je polje (i, j) već popunjeno
  if (A[i][j] != 0) {
    // ako je u pitanju poslednje polje, uspešno smo popunili ceo
    // sudoku
    if (rbr == n * n * n * n - 1)
      return true;
    // rekurzivno nastavljamo sa popunjavanjem
    return sudoku(A, rbr + 1);
  } else {
    // razmatramo sve moguće vrednosti koje možemo da upišemo na polje
    // (i, j)
    for (int k = 1; k <= n*n; k++) {
      // upisujeAo vrednost k
      A[i][j] = k;
      // ako time napravljen neki konflikt, nastavljamo popunjavanje
      // (pošto je polje popunjeno, na sledeće polje će se automatski
      // preći u rekurzivnom pozivu)
      // ako se sudoku uspešno popuni, prekidaAo dalju pretragu
      if (!konflikt(A, i, j))
        if (sudoku(A, rbr))
          return true;
    }
    // poništavamo vrednost upisanu na polje (i, j)
    A[i][j] = 0;
    // konstatujemo da ne postoji rešenje
    return false;
  }
}

int main() {
  vector<vector<int>> A(n * n);
  for (int i = 0; i < n * n; i++) {
    A[i].resize(n * n);
    string s;
    getline(cin, s);
    for (int j = 0; j < n * n; j++)
      A[i][j] = s[j] - '0';
  }

  if (!sudoku(A, 0))
    cout << "nema" << endl;
  else {
    for (int i = 0; i < n*n; i++) {
      for (int j = 0; j < n*n; j++)
```

```
        cout << A[i][j];
      cout << endl;
    }
  }

  return 0;
}
```

Задатак: Латински квадрати

Латински квадрат димензије $n \times n$ садржи бројеве од 1 до n распоређене тако су у свакој врсти и у свакој колони сви елементи различити. Напиши програм који дати делимично попуњени квадрат на све начине попуњава до латинског квадрата.

Улаз: Са стандардног улаза се учитава димензија n ($1 \leq n \leq 6$), а затим и елементи квадрата (празна поља су обележена цифром 0).

Излаз: На стандардни излаз исписати све латинске квадрате (иза сваког написати празну линију).

Пример

Улаз	*Излаз*
3	123
120	231
000	312
000	
	123
	312
	231

Решење

Задатак решавамо рекурзивном претрагом у дубину (слично као у задатку Судоку). Дефинишемо рекурзивну процедуру која ће да испише сва решења. Уз матрицу која се попуњава добија и редни број поља које треба попунити. Попуњавање креће од горњег левог угла и тече врсту по врсту, све док не попунимо целу матрицу (када је цела матрица попуњена, процедура је исписује, али се претрага након тога наставља да би се пронашла сва решења). Координате поља се веома једноставно могу одредити на основу његовог редног броја (слично као у задатку Шаховско поље). Ако је тренутно поље већ попуњено (јер су у старту нека поља већ попуњена), тада одмах прелазимо на наредно поље. У супротном проверавамо све могуће вредности за то поље. Након уписа вредности проверавамо да ли је тиме направљен неки конфликт тј. да ли се десило да је у истој врсти или у истој колони већ постојао број који је уписан. Ако јесте, претрагу прекидамо, а ако није, настављамо је даље, попуњавањем наредног поља.

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

bool konflikt(const vector<vector<int>>& A, int n, int i, int j) {
  // da li se A[i][j] nalazi već u koloni j
  for (int k = 0; k < n; k++)
    if (k != i && A[i][j] == A[k][j])
      return true;

  // da li se A[i][j] nalazi već u vrsti i
  for (int k = 0; k < n; k++)
    if (k != j && A[i][j] == A[i][k])
      return true;

  // ne postoji konflikt
  return false;
```

```
}

void latinskiKvadrati(vector<vector<int>>& A, int n, int rbr = 0) {
  int i = rbr / n, j = rbr % n;
  // ako je polje (i, j) već popunjeno
  if (A[i][j] != 0) {
    // ako je u pitanju poslednje polje, uspešno smo popunili ceo
    // sudoku
    if (rbr == n * n - 1) {
      for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
          cout << A[i][j];
        cout << '\n';
      }
      cout << '\n';
    } else {
      // rekurzivno nastavljamo sa popunjavanjem
      latinskiKvadrati(A, n, rbr + 1);
    }
  } else {
    // razmatramo sve moguće vrednosti koje možemo da upišemo na polje
    // (i, j)
    for (int k = 1; k <= n; k++) {
      // upisujemo vrednost k
      A[i][j] = k;
      // ako time napravljen neki konflikt, nastavljamo popunjavanje
      // (pošto je polje popunjeno, na sledeće polje će se automatski
      // preći u rekurzivnom pozivu)
      // ako se sudoku uspešno popuni, prekidamo dalju pretragu
      if (!konflikt(A, n, i, j))
        latinskiKvadrati(A, n, rbr);
    }
    // poništavamo vrednost upisanu na polje (i, j)
    A[i][j] = 0;
  }
}


int main() {
  int n;
  cin >> n >> ws;
  vector<vector<int>> A(n);
  for (int i = 0; i < n; i++) {
    A[i].resize(n);
    string s;
    getline(cin, s);
    for (int j = 0; j < n; j++)
      A[i][j] = s[j] - '0';
  }

  latinskiKvadrati(A, n);

  return 0;
}
```

Задатак: Скакачева тура

Напиши програм који одређује све могуће обиласке шаховске табле дате димензије скакачем. Скакач креће из горњег левог угла табле, у сваком кораку се креће у облику ћириличког слова Г, два поља хоризонтално и једно вертикално или два поља вертикално и једно хоризонтално и током обиласка свако поље на табли посећује тачно једном.

Улаз: Са стандардног улаза се уносе димензије табле $m \times n$ ($2 \leq 8$).

Излаз: На стандардни излаз исписати све могуће редоследе обиласка поља (у произвољном редоследу). Бројање поља започети од 1. Сваки број исписати са два карактера (испред једноцифрених додати размак) и иза сваког броја одштампати размак. Након сваког успешног обиласка исписати и један празан ред.

Пример

Улаз *Излаз*

```
3 4        1  4  7 10
           8 11  2  5
           3  6  9 12

           1  4  7 10
          12  9  2  5
           3  6 11  8
```

Решење

Задатак се може решити претрагом са повратком (бектрекингом), по узору на технике које смо користили приликом генерисања комбинаторних објеката. Парцијално решење у сваком кораку проширујемо тако што одређујемо све могуће начине да скакач пређе са текућег на наредно поље. За то је потребно да знамо које је поље већ обиђено, а које је слободно. Пошто ћемо желети да решења прикажемо у неком прегледном облику, можемо одржавати матрицу у којој ћемо на непосећеним пољима чувати нуле, а на посећеним пољима редне бројеве потеза када је то поље посећено. Током обиласка одржаваћемо и редни број тренутног потеза. Дефинисаћемо рекурзивну функцију чији је задатак да прошири започети обилазак тиме што ће се наредни потез чији је редни број дат одиграти на поље чије су координате такође дате. Претпоставићемо да ће функција бити позивана само ако смо сигурни да тај потез може бити одигран. Након одигравања потеза, провераваћемо да ли је табла комплетно попуњена (то можемо једноставно открити на основу димензија табле и редног броја одиграног потеза) и ако јесте, исписиваћемо пронађено решење. Ако није, бираћемо наредни потез тако што анализирамо све могуће потезе скакача, проверавати који је могуће одиграти (који води на неко непопуњено поље у оквирима табле) и позиваћемо рекурзивно функцију да настави обилазак од тог поља.

```
#include <iostream>
#include <iomanip>
#include <vector>

using namespace std;

int V, K;

vector<vector<int>> potezi =
  {{-1, -2}, {-1, 2}, {1, -2}, {1, 2}, {-2, -1}, {-2, 1}, {2, -1}, {2, 1}};

void ispisi(const vector<vector<int>>& tabla) {
  for (int v = 0; v < V; v++) {
    for (int k = 0; k < K; k++)
      cout << setw(2) << tabla[v][k] << " ";
    cout << endl;
  }
  cout << endl;
}

void obidjiTabluSkakacem(vector<vector<int>>& tabla, int v, int k, int potez) {
  tabla[v][k] = potez;
```

```
  if (potez == V*K)
    ispisi(tabla);
  for (auto& p : potezi) {
    int vn = v + p[0], kn = k + p[1];
    if (0 <= vn && vn < V &&
        0 <= kn && kn < K &&
        tabla[vn][kn] == 0) {
      obidjiTabluSkakacem(tabla, vn, kn, potez+1);
    }
  }
  tabla[v][k] = 0;
}

int main() {
  cin >> V >> K;
  vector<vector<int>> tabla(V);
  for (int i = 0; i < V; i++)
    tabla[i].resize(K, 0);
  obidjiTabluSkakacem(tabla, 0, 0, 1);
  return 0;
}
```

Задатак: Сви збирови обиласка матрице

Напиши програм који одређује све могуће збирове бројева који се могу добити тако што се обилазе поља матрице од горњег-левог угла, у сваком кораку се прелази доле или десно све док се не стигне до поља у доњем-десном углу.

Улаз: Са стандардног улаза се уноси број n ($3 \leq n \leq 10$), а затим у наредних n редова врсте матрице (елементи сваке врсте су раздвојени размацима).

Излаз: На стандардни излаз исписати све могуће збирове, раздвојене размацима, сваки у посебном реду.

Пример

Улаз	*Излаз*
3 10 7 7 7 7 5 4 7 2	30 31 33

Решење

Задатак можемо решити исцрпним испитивањем свих путања. По узору на листање свих варијација Све варијације градимо рекурзивну функцију која прима позицију тренутног поља и збир бројева на свим пољима којима се прошло док се није стигло до тренутног поља. Збир увећавамо за број на тренутном пољу. Ако је тренутно поље оно у доњем десном углу, онда смо стигли до краја и тренутни збир додајемо у скуп свих збирова. У супротном, вршимо рекурзивни позив у ком прелазимо на десно поље (ако већ нисмо у последњој колони) и рекурзивни позив у ком прелазимо на доње поље (ако већ нисмо у последњом врсти). У језику C++ скуп можемо представити помоћу библиотечке колекције `set`.

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

void pokupiZbirove(const vector<vector<int>>& M, int n, int i, int j, int zbir,
                   set<int>& zbirovi) {
  zbir += M[i][j];
  if (i == n-1 && j == n-1)
    zbirovi.insert(zbir);
```

```
  else {
    if (i < n-1)
      pokupiZbirove(M, n, i+1, j, zbir, zbirovi);
    if (j < n-1)
      pokupiZbirove(M, n, i, j+1, zbir, zbirovi);
  }
}

void pokupiZbirove(const vector<vector<int>>& M, int n, set<int>& zbirovi) {
  pokupiZbirove(M, n, 0, 0, 0, zbirovi);
}

int main() {
  int n;
  cin >> n;
  vector<vector<int>> M(n);
  for (int i = 0; i < n; i++) {
    M[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> M[i][j];
  }
  set<int> zbirovi;
  pokupiZbirove(M, n, zbirovi);
  for (auto zbir : zbirovi)
    cout << zbir << " ";
  cout << endl;
  return 0;
}
```

Задатак: Број поднизова датог збира

Напиши програм који одређује колико поднизова (не обавезно узастопних елемената) датог низа позитивних бројева има збир једнак датом броју.

Улаз: Са стандардног улаза се учитава број $1 \leq n \leq 30$, а затим у наредном реду n позитивних реалних бројева раздвојених размацима.

Излаз: На стандардни излаз исписати тражени број поднизова (два реална броја се могу сматрати једнакима ако се разликују за мање од 10^{-5}).

Пример

Улаз	*Излаз*
4 3.2 5.7 9.4 6.9 12.6	2

Решење

Једна могућност је да се задатак реши грубом силом, тј. да се наброје сви поднизови и да се за сваки од њих провери да ли му је збир једнак траженом. Набрајање поднизова се може постићи на било који од начина приказаних у задатку Сви подскупови. Једна могућност имплементације подразумева да чувамо елементе подниза, тренутни број елемената подниза и параметар који одређује који део скупа је још потребно обрадити (то може бити дужина префикса низа који још није обрађен). Када се тај префикс испразни, проверавамо текући подниз. У супротном рекурзивно разматрамо могућност да последњи елемент префикса није или јесте укључен у подниз, уклањајући у оба случаја тај последњи елемент из префикса (тако што се смањује његова дужина).

```
#include <iostream>
#include <vector>
#include <cmath>
```

```cpp
using namespace std;

const double EPS = 0.00001;

// racuna se broj podnizova koji elementima niza na pozicijama [k, n)
// nastavljaju dati podniz duzine k i koji imaju dati zbir
int brojPodnizovaDatogZbira(const vector<double>& niz, int n,
                            double ciljniZbir,
                            vector<double>& podniz, int k) {
  // u nizu nema preostalih elemenata, pa je trenutni podniz jedini kandidat
  if (n == 0) {
    // racunamo zbir trenutnog podniza
    double zbirPodniza = 0.0;
    for (int i = 0; i < k; i++)
      zbirPodniza += podniz[i];
    // proveravamo da li je jednak ciljnom zbiru
    if (abs(zbirPodniza - ciljniZbir) < EPS)
      return 1;
    else
      return 0;
  } else {
    // broj podnizova bez ukljucenog poslednjeg elementa niza
    int broj = 0;
    broj += brojPodnizovaDatogZbira(niz, n-1, ciljniZbir, podniz, k);
    // broj podnizova sa ukljucenim poslednjim elementom niza
    podniz[k] = niz[n-1];
    broj += brojPodnizovaDatogZbira(niz, n-1, ciljniZbir, podniz, k+1);
    return broj;
  }
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  vector<double> podniz(n);
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, n, ciljniZbir, podniz, 0);
}

int main() {
  // ucitavamo niz
  int n;
  cin >> n;
  vector<double> niz(n);
  for (int i = 0; i < n; i++)
    cin >> niz[i];
  // ciljni zbir podniza
  double ciljniZbir;
  cin >> ciljniZbir;

  // izracunavamo i ispisujemo trazeni broj podnizova
  cout << brojPodnizovaDatogZbira(niz, ciljniZbir) << endl;

  return 0;
}
```

Друга могућност имплементације претраге грубом силом је да као параметар рекурзивне функције прослеђујемо тренутни циљни збир тј. разлику између траженог збира и збира елемената тренутно укључених у

подниз. Сам подниз није неопходно одржавати. Ако је циљни збир једнак нули, то значи да је збир тренутног подниза једнак траженом и да смо нашли један задовољавајући подниз. У супротном, ако у скупу нема преосталих елемената, тада знамо да није могуће направити подниз траженог збира. У супротном уклањамо тренутни елемент из низа и разматрамо могућност да се он укључи и могућност да се не укључи у подниз. У првом случају умањујемо циљни збир за вредност тог елемента, а у другом циљни збир остаје непромењен.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

const double EPS = 0.00001;

// funkcija odredjuje broj podnizova niza odredjenog elementima na
// pozicijama [k, n) takvih da je zbir elemenata podniza jednak ciljnom zbiru
int brojPodnizovaDatogZbira(const vector<double>& niz, double ciljniZbir, int k) {
  // jedino prazan niz ima zbir nula
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
  if (k == niz.size())
    return 0;

  // posebno brojimo podnizove koji ukljucuju niz[k] i one koji ne ukljucuju
  // niz[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k], k+1) +
         brojPodnizovaDatogZbira(niz, ciljniZbir, k+1);
}

int brojPodnizovaDatogZbira(const vector<double>& niz, double ciljniZbir) {
  // brojimo podnizove niza odredjenog elementima na pozicijama [0, n)
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, 0);
}

int main() {
  // ucitavamo niz
  int n;
  cin >> n;
  vector<double> niz(n);
  for (int i = 0; i < n; i++)
    cin >> niz[i];
  // ucitavamo ciljni zbir
  double ciljniZbir;
  cin >> ciljniZbir;

  // izracunavamo i ispisujemo trazeni broj podnizova
  cout << brojPodnizovaDatogZbira(niz, ciljniZbir) << endl;

  return 0;
}
```

Ефикаснија решења од решења грубом силом се могу добити применом различитих одсецања. Кључна ствар је да одредимо интервал у коме могу лежати збирови свих поднизова преосталих елемената низа. Пошто су сви елементи позитивни, најмања могућа вредност збира подниза је нула (у случају празног низа), док је највећа могућа вредност збира подниза једнака збиру свих елемената низа. Дакле, ако је циљни збир строго мањи од нуле или строго већи од збира свих елемената преосталог дела низа, тада не постоји ни један подниз

чији је збир једнак циљном. Уместо да збир свих елемената низа рачунамо изнова у сваком рекурзивном позиву, можемо приметити да се у сваком наредном рекурзивном позиву низ само може смањити за један елемент, па се збир може рачунати инкрементално, умањивањем током рекурзије збира полазног низа за елементе уклоњене из низа.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

const double EPS = 0.00001;

// racuna se broj podnizova elemenata niza na pozicijama [k, n) koji
// imaju dati zbir, pri cemu se zna da je zbir tih elemenata jednak
// zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz, double ciljniZbir,
                            double zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan element
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
  if (k == niz.size())
    return 0;

  // posto su svi brojevi pozitivni, nije moguce dobiti negativan ciljni zbir
  if (ciljniZbir + EPS < 0)
    return 0;

  // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog zbira,
  // pa nema podnizova koji bi dali ciljni zbir
  if (zbirPreostalih + EPS < ciljniZbir)
    return 0;

  // broj podnizova u kojima ucestvuje element a[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                                 zbirPreostalih - niz[k], k+1) +
    // broj podnizova u kojima ne ucestvuje element a[k]
    brojPodnizovaDatogZbira(niz, ciljniZbir,
                            zbirPreostalih - niz[k], k+1);
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  // izracunavamo zbir elemenata niza
  double zbirNiza = 0;
  for (int i = 0; i < n; i++)
    zbirNiza += niz[i];
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}


int main() {
  // ucitavamo niz
  int n;
```

```
  cin >> n;
  vector<double> niz(n);
  for (int i = 0; i < n; i++)
    cin >> niz[i];
  // ucitavamo ciljni zbir
  double ciljniZbir;
  cin >> ciljniZbir;

  // izracunavamo i ispisujemo trazeni broj podnizova
  cout << brojPodnizovaDatogZbira(niz, ciljniZbir) << endl;

  return 0;
}
```

Још једно могуће одсецање се може извршити када се установи да је најмањи од преосталих бројева у низу већи од циљног збира. Ако је тај циљни збир позитиван, тада није могуће достићи га (јер празан подниз има збир нула, а било који непразан подниз има збир већи или једнак од тог минималног елемента). Минимални елемент преосталог дела низа је једноставно одредити ако се низ сортира (што можемо урадити пре почетка претраге).

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

const double EPS = 0.00001;

// racuna se broj podnizova elemenata niza na pozicijama [k, n) koji
// imaju dati zbir, pri cemu se zna da je zbir tih elemenata jednak
// zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz, double ciljniZbir,
                            double zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan element
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
  if (k == niz.size())
    return 0;

  // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog zbira,
  // pa nema podnizova koji bi dali ciljni zbir
  if (zbirPreostalih + EPS < ciljniZbir)
    return 0;

  // vec uzimanje najmanjeg elementa prevazilazi ciljni zbir, pa
  // nema podnizova koji bi dali ciljni zbir
  if (niz[k] > ciljniZbir + EPS)
    return 0;

        // broj podnizova u kojima ucestvuje element a[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                                 zbirPreostalih - niz[k], k+1) +
        // broj podnizova u kojima ne ucestvuje element a[k]
        brojPodnizovaDatogZbira(niz, ciljniZbir,
                                zbirPreostalih - niz[k], k+1);
}
```

```cpp
// funkcija racuna koliko podnizova datog niza ima zbir jednak ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  // sortiramo elemente niza neopadajuce
  sort(begin(niz), end(niz));
  // izracunavamo zbir elemenata niza
  double zbirNiza = 0;
  for (int i = 0; i < n; i++)
    zbirNiza += niz[i];
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}


int main() {
  // ucitavamo niz
  int n;
  cin >> n;
  vector<double> niz(n);
  for (int i = 0; i < n; i++)
    cin >> niz[i];
  // ciljni zbir podniza
  double ciljniZbir;
  cin >> ciljniZbir;

  // izracunavamo i ispisujemo trazeni broj podnizova
  cout << brojPodnizovaDatogZbira(niz, ciljniZbir) << endl;

  return 0;
}
```

Задатак: Мерење са n тегова

Дато је n тегова, за сваки тег позната је његова маса. Датим теговима треба измерити масу S тако да се укупна маса изабраних тегова најмање разликује од S. Написати програм којим се одређује минимална разлика при таквом мерењу.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 10$). Следећих n линија садрже реалне бројеве, сваки у посебној линији, који представљају масе датих тегова. Последња линија стандардног улаза садржи реалан број S који представља масу коју меримо.

Излаз: На стандардном излазу приказати у једној линији минималну разлику добијену при мерењу, разлику приказати на две децимале.

Пример

Улаз	*Излаз*
5 2.3 1 0.5 2 0.25 4	0.05

Решење

Решење грубом силом подразумева да се испитају сви могући подскупови датог скупа тегова. Генерисање свих подскупова смо разматрали у задатку Сви подскупови. На пример, набрајање можемо остварити помоћу

функције која проналази лексикографски следећи подскуп. За сваки генерисани подскуп израчунавамо масу тегова, рачунамо одступање од жељене масе и ако је оно мање од тренутно најмањег одступања, ажурирамо минимум. Најмање одступање је могуће иницијализовати на жељену масу, јер масу 0 увек можемо добити помоћу празног скупа тегова. Рецимо да бисмо заједно са подскупом могли одржавати и масу предмета у подскупу и приликом проналажења наредног подскупа ажурирати ту масу, чиме би се програм мало убрзао.

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <vector>

using namespace std;

// funkcija pronalazi sledeci podskup (leksikografski sledecu
// varijaciju elemenata false i true)
bool sledeciPodskup(vector<bool>& uSkupu) {
  int i;
  for (i = uSkupu.size() - 1; i >= 0 && uSkupu[i]; i--)
    uSkupu[i] = false;

  if (i < 0)
    return false;

  uSkupu[i] = true;
  return true;
}

int main() {
  // ucitavamo podatke o tegovima
  int n;
  cin >> n;
  vector<double> tegovi(n);
  for(int i = 0; i < n; i++)
    cin >> tegovi[i];
  // ukjucujemo ciljnu masu
  double ciljnaMasa;
  cin >> ciljnaMasa;

  // krecemo od praznog skupa tegova
  vector<bool> uSkupu(n, false);
  double minRazlika = ciljnaMasa;
  do {
    // izracunavamo masu tegova u tekucem skupu
    double tekucaMasa = 0;
    for(int i = 0; i < n; i++)
      if (uSkupu[i])
        tekucaMasa += tegovi[i];
    // azuriramo minimalnu razliku, ako je to potrebno
    double tekucaRazlika = abs(ciljnaMasa - tekucaMasa);
    if (tekucaRazlika < minRazlika)
      minRazlika = tekucaRazlika;
  } while (sledeciPodskup(uSkupu));

  // ispisujemo najmanju razliku
  cout << fixed << setprecision(2) << showpoint
       << minRazlika << endl;
  return 0;
}
```

Рекурзивну функцију која генерише све подскупове можемо модификовати тако да враћа вредност најмањег одступања од циљне тежине. Подсетимо се, функција прима текући подскуп елемената низа са позиција из интервала $[0, i)$ и на све начине га проширује елементима низа из интервала $[i, n)$. Када је $i = n$, генерисан је подскуп целог низа, израчунава се његово одступање и враћа се резултат (то је једино, па уједно и најмање одступање). У супротном се разматрају две могућности: елемент на позицији i се или додаје у подскуп или се из подскупа изоставља. Вршимо два рекурзивна позива и мање од њихова два одступања нам даје тражено минимално одступање (то је најмање одступање за све подскупове који садрже елементе са позиција $[0, i)$ који су тренутно одабрани). Централни позив се врши за $i = 0$ (у почетку ништа није одабрано и сви елементи тегови нам на располагању). Приметимо да заправо није неопходно да знамо који су тачно тегови тренутно одабрани, већ само њихову укупну масу.

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <algorithm>

using namespace std;

double merenje(const vector<double>& tegovi, double ciljnaMasa,
               int i, double tekucaMasa) {
  // nemamo vise tegova koje mozemo uzimati
  if (i == tegovi.size())
    // najmanja razlika je odredjena tekucom masom ukljucenih tegova
    return abs(ciljnaMasa - tekucaMasa);

  // gledamo bolju mogućnost od one kada preskacemo teg na poziciji i
  // i one kada uključimo teg na poziciji i
  return min(merenje(tegovi, ciljnaMasa, i+1, tekucaMasa),
             merenje(tegovi, ciljnaMasa, i+1, tekucaMasa + tegovi[i]));
}

double merenje(const vector<double>& tegovi, double ciljnaMasa) {
  // krecemo od pozicije 0 i praznog skupa ukljucenih tegova
  return merenje(tegovi, ciljnaMasa, 0, 0.0);
}

int main() {
  int n;
  cin >> n;
  vector<double> tegovi(n);
  for (int i = 0; i < n; i++)
    cin >> tegovi[i];
  double ciljnaMasa;
  cin >> ciljnaMasa;
  cout << fixed << setprecision(2) << showpoint
       << merenje(tegovi, ciljnaMasa) << endl;
  return 0;
}
```

Ефикасније решење можемо добити ако током исцрпне претраге применимо одсецање. На пример, након што одредимо најмању разлику r без укључивања текућег тега и ако додавање текућег тега даје масу која је већа од циљане и чија је разлика од циљане већа од r, тада додавањем текућег тега није могуће добити мању разлику (јер ће додавање наредних тегова моћи само да повећа масу).

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <algorithm>
```

```
using namespace std;

// funkcija odredjuje najmanju razliku izmedju ciljne mase koju treba izmeriti i
// mase koja se može postici pomoću tegova iz intervala [k, n) cija je ukupna
// masa jednaka broju preostalaMasa
double merenje(const vector<double>& tegovi, double ciljnaMasa,
               int k, double tekucaMasa, double preostalaMasa) {
  // nemamo vise tegova koje mozemo uzimati
  if (k == tegovi.size())
    // najmanja razlika je odredjena tekucom masom ukljucenih tegova
    return abs(ciljnaMasa - tekucaMasa);

  // preskacemo teg na poziciji k
  double minRazlika = merenje(tegovi, ciljnaMasa,
                              k+1, tekucaMasa, preostalaMasa - tegovi[k]);
  // ako ukljucivanje tega na poziciji i ima sanse da da manju razliku
  // od najmanje
  if (tekucaMasa + preostalaMasa > ciljnaMasa - minRazlika &&
      tekucaMasa + tegovi[k] < ciljnaMasa + minRazlika)
    // ukljucujemo ga i gledamo da li je to popravilo najmanju razliku
    minRazlika = min(minRazlika,
                     merenje(tegovi, ciljnaMasa,
                             k+1, tekucaMasa + tegovi[k], preostalaMasa - tegovi[k]));
  // vracamo najmanju razliku
  return minRazlika;
}

double merenje(const vector<double>& tegovi, double ciljnaMasa) {
  double ukupnaMasa = 0.0;
  for (int i = 0; i < tegovi.size(); i++)
    ukupnaMasa += tegovi[i];
  // krecemo od pozicije 0 i praznog skupa ukljucenih tegova
  return merenje(tegovi, ciljnaMasa, 0, 0.0, ukupnaMasa);
}

int main() {
  int n;
  cin >> n;
  vector<double> tegovi(n);
  for (int i = 0; i < n; i++)
    cin >> tegovi[i];
  double ciljnaMasa;
  cin >> ciljnaMasa;
  cout << fixed << setprecision(2) << showpoint
       << merenje(tegovi, ciljnaMasa) << endl;
  return 0;
}
```

Рецимо и да имплементацију можемо направити мало другачије. Уместо функције која враћа најмању вредност одступања, можемо направити процедуру (функцију без повратне вредности) која ажурира глобалну променљиву. Ипак, резоновање о програму је много једноставније ако се не користе глобалне променљиве.

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;
```

```cpp
// maksimalni broj tegova
const int MAX_TEGOVA = 30;

// broj tegova
int n;
// tezine tegova
double tegovi[MAX_TEGOVA];
// masa koju je potrebno izmeriti
double ciljnaMasa;
// najmanja razlika
double minRazlika;

void merenje(double tekucaMasa, int i) {
  // razmotrili smo svih n tegova
  if (i == n) {
    // azuriramo razliku ako je potrebno
    double razlika = abs(ciljnaMasa - tekucaMasa);
    if (razlika < minRazlika)
      minRazlika = razlika;
    return;
  }
  // preskacemo tekuci teg
  merenje(tekucaMasa, i+1);
  // ukljucujemo tekuci teg (osim ako se njegovim ukljucivanjem ne premasi
  // ciljna masa za vise od trenutne minimalne razlike)
  if (tekucaMasa + tegovi[i] < ciljnaMasa + minRazlika)
    merenje(tekucaMasa + tegovi[i], i+1);
}

int main() {
  // ucitavamo podatke o tegovima
  cin >> n;
  for (int i = 0; i < n; i++)
    cin >> tegovi[i];
  // ucitavamo ciljnu masu
  cin >> ciljnaMasa;

  // krecemo od toga da smo izmerili 0.0
  minRazlika = ciljnaMasa;
  merenje(0.0, 0);

  // ispisujemo rezultat
  cout << fixed << setprecision(2) << showpoint
       << minRazlika << endl;
  return 0;
}
```

Задатак: Мерење са n врста тегова

Дато је n врста тегова, за сваки тег позната је његова маса (реалан број) и колико тегова те врсте имамо на располагању (природан број). Датим теговима треба измерити масу S са тачношћу од 2 децимале. Написати програм којим се проверава да ли је то могуће.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 10$). Свака од следећих n линија садржи реалан и природан број одвојени једним бланко карактером, који редом представљају масу тега и број тегова те масе које имамо на располагању. Последња линија стандардног улаза садржи реалан број S који представља масу коју меримо.

Излаз: На стандардном излазу приказати поруку `da` ако је тражено мерење могуће извршити иначе приказати

поруку ne.

Пример

Улаз	*Излаз*
3 0.255 4 1.041 3 2.00 2 1.80	da

Решење

Решење грубом силом подразумева да се испитају све могућности и да се утврди да ли је нека од њих задовољавајућа. Свака могућност у овом случају је нека варијација са понављањем, при чему је ограничење на свакој позицији у варијацији другачије (на свакој позицији тј. од сваке врсте тегова можемо узети највише онолико тегова колико постоји тегова те врсте). Генерисање свих варијација са понављањем смо разматрали у задатку Све варијације. Крећемо од варијације у којој није узет ни један тег, у сваком кораку за текућу варијацију рачунамо измерену масу и проверавамо да ли је она једнака траженој (до на задату прецизност 10^{-2}). Ако јесте, тражено мерење постоји, па петљу можемо прекинути, а ако није, прелазимо на наредну варијацију, све док таква постоји. Приликом генерисања наредне варијације крећемо се од краја (последње врсте тегова) и тражимо врсту тегова која није достигла максимум, тј. где је број узетих тегова у варијацији мањи од броја расположивих тегова те врсте. Ако таква врста тегова не постоји, нема ни следеће варијације, а у супротном узимамо још један тег те врсте и уклањамо све тегове наредних врста.

```
#include <iostream>
#include <algorithm>
#include <cmath>

using namespace std;

// preciznost zadata u tekstu zadatka
const double preciznost = 0.01;
// maksimalni broj vrsta tegova zadat teksom zadatka
const int MAX_BROJ_TEGOVA = 10;

// pronalazi sledecu varijaciju sa ponavljanjem broja uzetih tegova,
// pod pretpostavkom da imamo ogranicenu kolicinu svakog od tegova
bool sledecaVarijacija(int brojUzetihTegova[], int brojTegova[], int n) {
  int i = n-1;
  while(i >= 0 && brojUzetihTegova[i] == brojTegova[i]) {
    brojUzetihTegova[i] = 0;
    i--;
  }
  if (i < 0)
    return false;
  brojUzetihTegova[i]++;
  return true;
}

int main() {
  // ucitavamo podatke o tegovima
  int n;
  cin >> n;
  double masaTega[MAX_BROJ_TEGOVA];
  int brojTegova[MAX_BROJ_TEGOVA];
  for(int i = 0; i < n; i++)
    cin >> masaTega[i] >> brojTegova[i];
  // ucitavamo ciljnu masu
  double ciljnaMasa;
  cin >> ciljnaMasa;
```

```
  // krecemo od kombinacije u kojoj nismo uzeli ni jedan teg
  int brojUzetihTegova[10];
  for(int i = 0; i < n; i++)
    brojUzetihTegova[i] = 0;

  // belezimo da li smo pronasli uspesno merenje
  bool moze = false;
  do {
    // racunamo masu svih uzetih tegova
    double masaUzetihTegova = 0.0;
    for (int i = 0; i < n; i++)
      masaUzetihTegova += masaTega[i] * brojUzetihTegova[i];
    // ako je ona jednaka ciljnoj masi (do na dozvoljenu preciznost),
    // merenje je moguce
    if (abs(ciljnaMasa - masaUzetihTegova) < preciznost)
      moze = true;
  } while(sledecaVarijacija(brojUzetihTegova, brojTegova, n) && !moze);

  // ispisujemo rezultat
  if (moze)
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Све варијације можемо генерисати и рекурзивном функцијом и тада током генерисања можемо извршити и одсецање којим проверу чинимо ефикаснијом. Дефинишемо функцију која на основу одабраних тегова свих врста из интервала $[0, i)$ покушава да изврши мерење додајући тегове чије су врсте из интервала $[i, n)$, при чему се након додавања тегова врсте i циљна маса смањује. Ако је при уласку у рекурзију циљна маса једнака нули (до на допуштену прецизност), значи да је почетна маса успешно измерена помоћу тегова чије су врсте из интервала $[i, n)$ и функција може да констатује да је мерење могуће. У супротном, ако је $i = n$ не постоје тегови који би се додали и мерење није могуће. Такође, ако је циљна маса негативна (и мања од допуштене толеранције) мерење није могуће (јер се додавањем тегова циљна маса не може повећати). У супротном разматрамо разне могућности за додавање тегова врсте i. У петљи проверавамо број тих тегова од 0 па све до укупног броја тих тегова. Ако током петље установимо да се додавањем тих тегова превазишла циљна маса (укључујући и допуштену толеранцију) петљу можемо прекинути. У сваком кораку петље вршимо рекурзивни позив и ако било који од њих врати вредност `true` констатујемо да је пронађено успешно мерење. Ако се петља изврши и сви рекурзивни позиви врате `false`, тада мерење није могуће.

```
#include <iostream>
#include <algorithm>
#include <cmath>

using namespace std;

// maksimalni broj vrsta tegova zadat teksom zadatka
const int MAX_BROJ_TEGOVA = 10;
// preciznost zadata u tekstu zadatka
const double preciznost = 0.01;

// broj vrsta tegova
int n;
// mase i kolicine svake vrste tega
double masaTega[MAX_BROJ_TEGOVA];
int brojTegova[MAX_BROJ_TEGOVA];

// odredjuje da li je moguce izmeriti preostalu ciljnu tezinu
// bez prvih i vrsta tegova
```

```
bool merenje(double ciljnaMasa, int i) {
  // ako je masa izmerena pomocu prethodnih tegova (do na trazenu
  // preciznost) uspesno merenje je moguce
  if (abs(ciljnaMasa) < preciznost)
    return true;
  // u suprotnom, ako nema vise tegova, merenje nije moguce
  if (i == n)
    return false;
  // negativnu masu ne mozemo meriti
  // (masa prethodno odabranih tegova je veca od ciljne tezine)
  if (ciljnaMasa < 0)
    return false;
  // isprobavamo sve mogucnosti za tegove vrste broj i stajemo kada
  // nema vise tegova te vrste ili kada je broj uzetih tegova te vrste
  // vec veci od ciljne mase (do na trazenu preciznost), pri cemu
  // ciljnu masu smanjujemo za uzete tegove
  for (int k = 0; k <= brojTegova[i] &&
                  k*masaTega[i] <= ciljnaMasa + preciznost; k++)
    if (merenje(ciljnaMasa - k*masaTega[i], i+1))
      return true;

  // nismo uspeli da izmerimo ciljnu masu
  return false;
}

int main() {
  // ucitavamo podatke o tegovima
  cin >> n;
  for(int i = 0; i < n; i++)
    cin >> masaTega[i] >> brojTegova[i];
  // ucitavamo ciljnu masu
  double ciljnaMasa;
  cin >> ciljnaMasa;

  // proveravamo da li je merenje ciljne mase moguce (pomocu svih
  // tegova)
  if (merenje(ciljnaMasa, 0))
    cout << "da" << endl;
  else
    cout << "ne" << endl;

  return 0;
}
```

Задатак: Број израза дате вредности

Дат је стринг s који садржи само цифре (0, …, 9) и природан број x. Написати програм којим се одређује број израза који се могу добити уметањем оператора $+$, $-$ и $\cdot$ у стрингу s тако да је вредност добијеног израза једнака x. При том, сваки операнд у том изразу мора да буде исправно записан природан број (вишецифрени бројеви не смеју да почињу нулом).

Улаз: У првој линији стандардног улаза налази се стринг s, друга линија садржи природан број x.

Излаз: На стандарном излазу у једној линији приказати број тражених израза.

Пример

Улаз	*Излаз*
1009 10	8

Решење

Генерисање свих израза

Један од најдиректнијих начина да се реши задатак је да се генеришу сви могући описани изрази, да се израчуна вредност свакога и да се преброје они који имају тражену вредност.

Генерисање свих израза је слично поступку генерисања свих варијација (који смо приказали у задатку Све варијације). Поступно градимо ниску карактера која садржи израз. Генерисање почињемо од израза који садржи само прву цифру оригиналне ниске. У сваком кораку текући израз проширујемо текућим карактером оригиналне ниске цифара и за свако такво проширење рекурзивно настављамо исти поступак све док се карактери оригиналне ниске не исцрпе. Проширење текућег израза цифром можемо остварити на 4 начина: само дописујући ту цифру или дописујући ту цифру након сваког од 3 допуштена оператора (на пример, израз `32+4` можемо проширити цифром `5` тако да добијемо изразе `32+45`, `32+4+5`, `32+4-5` и `32+4*5`). Први случај (дописивање цифре без оператора) није допуштен ако се текући израз завршава једноцифреним бројем `0`.

Рачунање вредности израза је већ приказано у задатку Израз у коме нема заграда.

```
#include <iostream>
#include <string>
#include <cctype>

using namespace std;

long long vrednost(const string& s) {
  long long rezultat = 0;
  int znakTekucegSabirka = 1;
  long long tekuciSabirak = 1;
  long long tekuciBroj = 0;
  for (int i = 0; i <= s.length(); i++)
    if (i < s.length() && isdigit(s[i]))
      // procitali smo cifru
      // dodajemo je kao poslednju cifru tekuceg broja
      tekuciBroj = 10 * tekuciBroj + s[i] - '0';
    else {
      // dosli smo do nekog operatora ili kraja broja

      // tekuci broj je faktor tekuceg sabirka
      tekuciSabirak *= tekuciBroj;
      // zavrsili smo sa obradom tekuceg broja i priremamo se za citanje narednog
      tekuciBroj = 0;

      // ako smo stigli do kraja ili procitali aditivni operator
      // zavrsili smo obradu tekuceg sabirka
      if (i == s.length() || s[i] == '+' || s[i] == '-') {
        // rezultat uvecavamo ili umanjujemo za tekuci sabirak u
        // zavisnosti od ranije odredjenog znaka
        rezultat += znakTekucegSabirka * tekuciSabirak;
        if (i < s.length()) {
          // priremamo se za obradu narednog sabirka
          tekuciSabirak = 1;
          // njegov znak postavljamo u zavisnosti od operatora na koji
          // smo naisli
          znakTekucegSabirka = s[i] == '+' ? 1 : -1;
        }
      }
    }
  return rezultat;
}
```

```
// odredjuje broj nacina da se dobije vrednost x prosirivanjem izraza
// tekuciIzraz koriscenjem karaktera iz s pocevsi od pozicije poz
int brojNacina(const string& cifre, int poz, int x, string tekuciIzraz) {
  if (poz == cifre.length())
    // obradili smo sve karaktere niske s, pa se tekuci izraz ne moze
    // dalje prosirivati
    // broj nacina da se dobije x prosirivanjem tekuceg izraza je 1 ako
    // je vrednost tog izraza jednaka broju x, tj. 0 u suprotnom
    return vrednost(tekuciIzraz) == x ? 1 : 0;

  // broj nacina na koji mozemo prosiriti izraz tako da mu je vrednost x
  int ukupanBrojNacina = 0;

  // tekuci izraz prosirujemo cifrom s[poz]

  // ako se tekuci izraz ne zavrsava jednocifrenim brojem 0
  if (tekuciIzraz[tekuciIzraz.length() - 1] != '0' ||
      (tekuciIzraz.length() > 1 && isdigit(tekuciIzraz[tekuciIzraz.length() - 2])))
    // cifru samo dopisujemo na tekuci izraz
    ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz + cifre[poz]);

  // cifru dopisujemo iza svakog od dopustenih operatora
  ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz + "+" + cifre[poz]);
  ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz + "-" + cifre[poz]);
  ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz + "*" + cifre[poz]);
  return ukupanBrojNacina;
}

// broj nacina da se umetanjem operatora +, -, * u cifre date u nizu s
// dobije izraz cija je vrednost x
int brojNacina(const string& cifre, int x) {
  // tekuci izraz inicijalizujemo na prvu cifru
  return brojNacina(cifre, 1, x, string(1, cifre[0]));
}

int main() {
   string cifre;
   cin >> cifre;
   int ciljnaVrednost;
   cin >> ciljnaVrednost;
   cout << brojNacina(cifre, ciljnaVrednost) << endl;
   return 0;
}
```

Израчунавање вредности свих израза без њиховог генерисања

Претходно решење се може унапредити. Једна од мана је то што се у решењу манипулише са нискама које чувају текући израз, што може бити неефикасно. Друга, још упадљивија мана је то што пуно изграђених израза дели заједнички префикс чија се вредност непотребно израчунава много пута. Основа унапређења решења је то да се кроз рекурзију уместо ниске која представља текући израз шаље вредност текућег израз.

Први корак је да се одабере први број у изразу који градимо. Ако је прва цифра дате ниске нула, онда је једини начин да први број буде `0`. У супротном први број може бити било који непразан префикс наше ниске и тада се укупан број начина може добити сабирањем броја начина за сваки избор првог броја (на пример, ако је ниска `123` сабирамо колико има израза који почињу са `1`, колико са `12`, а колико са `123`).

Након одређивања првог броја крећемо да постављамо операторе. Ако је потребно да поставимо оператор на позицију `p` тада бирамо други операнд и то у делу ниске који почиње на позицији `p`. Њега бирамо на исти начин као и први број. Ако је цифра на позицији `p` нула, тада је једини начин да се тај операнд изабере баш та нула, а у супротном је могуће одабрати било који непразан префикс дела ниске који почиње на позицији

p. Када је одабран други операнд, вредност претходног израза треба проширити њиме и то тако да се између налази било који од три допуштена оператора. У случају оператора + и - вредност проширеног израза се директно ажурира. У случају оператора * вредност новог израза се добија тако што се од претходне вредности израза одузме вредност последњег сабирка а затим дода вредност тог сабирка помножена са вредношћу другог операнда. Због тога је уз вредност текућег израза кроз рекурзивне позиве потребно прослеђивати и вредност последњег урачунатог сабирка. Ово у потпуности одговара поступку спекулативног израчунавања вредности подизраза који смо описали у задатку Израз у коме нема заграда.

```
#include <iostream>
#include <string>

using namespace std;

// Neka je poznat izraz e + c, takav da mu je vrednost jednaka broju
// vrednost, a da je vrednost njegovog poslednjeg sabirka
// jednaka broju poslednjiSabirak. Funkcija izracunava broj nacina da
// se taj izraz prosiri ciframa niske s, pocevsi od pozicije poz, tako
// da prosireni izraz ima vrednost jednaku broju ciljnaVrednost.
int brojNacina(const string& cifre, int poz,
               long long vrednost, long long poslednjiSabirak,
               int ciljnaVrednost) {
  // ako nema vise karaktera izraz se ne moze prosiriti i proveravamo
  // da li je njegova vrednost jednaka ciljnoj
  if (poz == cifre.length())
    return vrednost == ciljnaVrednost ? 1 : 0;

  // ukupan broj nacina
  int ukupanBrojNacina = 0;
  // odredjujemo broj kojim prosirujemo izraz (tako sto izmedju izraza
  // i tog broja umecemo neki od dopustenih operatora)
  long long broj = 0;
  for (int i = poz; i < cifre.length(); i++) {
    // ako je cifre[poz] = 0, tada drugi operand moze biti samo 0, sto je
    // slucaj vec obradjen u prvoj iteraciji petlje, pa se petlja moze
    // prekinuti
    if (cifre[poz] == '0' && i > poz)
      break;

    // tekuci broj prosirujemo jos jednom cifrom
    broj = broj * 10 + cifre[i] - '0';

    // tako odredjen drugi operand kombinujemo sa izrazom pomocu
    // svakog od tri dopustena operatora, rekurzivno izracunavamo
    // odgovarajuca prosirenja tako dobijenih izraza i sve dobijene
    // ukupanBrojNacinae sabiramo
    ukupanBrojNacina += brojNacina(cifre, i+1, vrednost + broj, broj, ciljnaVrednost);
    ukupanBrojNacina += brojNacina(cifre, i+1, vrednost - broj, -broj, ciljnaVrednost);
    ukupanBrojNacina += brojNacina(cifre, i+1,
                                   vrednost - poslednjiSabirak + poslednjiSabirak * broj, poslednjiSabirak
                                   ciljnaVrednost);
  }
  // vracamo ukupan broj nacina
  return ukupanBrojNacina;
}

// broj nacina da se umetanjem operatora +, -, * izmedju cifara dobije
// izraz cija je vrednost jednaka broju ciljnaVrednost
int brojNacina(const string& cifre, int ciljnaVrednost) {
  // ukupan broj nacina da se napravi izraz sa ciljnom vrednoscu
```

```
  int ukupanBrojNacina = 0;
  // odredjujemo sve moguce prve clanove tog izraza
  long long broj = 0;
  for (int i = 0; i < cifre.length(); i++) {
    // ako izraz pocinje nulom, onda je jedini moguci prvi clan bas ta
    // nula - ako je ona obradjena u prethodnoj iteraciji, petlja se
    // moze zaustaviti
    if (cifre[0] == '0' && i > 0)
      break;

    // prosirujemo tekuci broj tekucom cifrom
    broj = broj * 10 + cifre[i] - '0';
    // izracunavamo broj nacina da se dobije ciljna vrednost
    // prosirivanjem izraza koji se sastoji samo od tog pocetnog broja
    ukupanBrojNacina += brojNacina(cifre, i + 1, broj, broj, ciljnaVrednost);
  }
  return ukupanBrojNacina;
}

int main() {
   string cifre;
   cin >> cifre;
   int ciljnaVrednost;
   cin >> ciljnaVrednost;
   cout << brojNacina(cifre, ciljnaVrednost) << endl;
   return 0;
}
```

Задатак: Дужина најдужег проходног пута

Дата је матрица $A_{v \times k}$ попуњења вредностима 0 и 1 (0 означава проходно поље а 1 препреку). Написати програм којим се одређује број корака на најдужем проходном путу од дате позиције (v_s,k_s) до дате позиције (v_c,k_c). Под једним кораком подразумева се прелазак са поља на једно од 4 суседна поља. Са сваког поља дозвољено је прећи на 4 суседна поља (горе, доле, лево и десно) ако су проходна. Сва поља на путу морају бити различита (на једно поље можемо стати највише једанпут). На стартном ни на циљном пољу се не налази препрека.

Улаз: Прва линија стандардног улаза садржи димензије матрице v и k ($1 < v, k \leq 10$), два природна броја одвојена једним размаком. Затим се на стандардном улазу налазе елементи матрице (у n линија по m бројева одвојених са по једним размаком). У последњој линији стандардног улаза налазе се 4 природна броја v_s, k_s, v_c, k_c ($0 \leq v_s, v_c < v, 0 \leq k_s, k_c < k$).

Излаз: На стандардном излазу приказати природан број који представља број корака на најдужем проходном путу од стартне позиције (v_s, k_s) до циљне позиције (v_c, k_c). Ако проходан пут не постоји приказати -1.

Пример

Улаз	*Излаз*
4 5 1 0 0 1 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 3 4	10

Решење

Задатак решавамо исцрпном рекурзивном претрагом. Рекурзивна функција прима матрицу која садржи позиције препрека, њену димензију, помоћну матрицу у којој се памти да ли је неко поље већ посећено (јер пут не сме два пута да посети једно те исто поље) и координате текућег и циљног поља. Ако је текуће поље једнако циљном, најдужи пут има дужину нула (не смемо да кренемо неким дужим путем, јер бисмо се тада

вратили на исто поље, што је забрањено). У супротном обележавамо да је текуће поље посећено, анализирамо њему четири суседна поља и ако су доступна (нису раније посећена и на њима није препрека) рекурзивно проналазимо дужину најдужег пута до циља. Дужина најдужег пута из текућег поља је онда за један већа од највеће дужине пута из неког од та четири суседна поља (разматрају се само они путеви који успешно стижу до циља).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// sa standardnog ulaza se ucitavaju dimenzije matrice, a zatim i sama matrica
// koja sadrzi vrednosti 0 i 1 (tumacimo ih kao logicke vrednosti)
void citajMatricu(vector<vector<bool>>& A, int& bv, int& bk) {
  cin >> bv >> bk;
  A.resize(bv);
  for (int i = 0; i < bv; i++) {
    A[i].resize(bk);
    for (int j = 0; j < bk; j++) {
      int x;
      cin >> x;
      A[i][j] = x != 0;
    }
  }
}

// duzina najduzeg puta u matrici A dimenzije bv x bk, gde su sa false
// oznacena mesta koja ne sadrze prepreku, ako se prelazi od tekuceg
// polja (v, k) i stize do ciljnog polja (vCilj, kCilj), pri cemu put
// ne sme da predje preko polja koja su obelezena sa true u metrici posecen
// ako put ne postoji, funkcija vraca -1
int najduziProhodanPut(const vector<vector<bool>>& A, int bv, int bk,
                       vector<vector<bool>>& posecen,
                       int v, int k, int vCilj, int kCilj) {
  // tekuce i ciljno polje se poklapaju
  if (v == vCilj && k == kCilj)
    return 0;

  // vrsimo pretragu 4 susedna polja
  posecen[v][k] = true;
  int maksDuzina = -1;
  int dv[] = {1, -1, 0, 0};
  int dk[] = {0, 0, 1, -1};
  for (int i = 0; i < 4; i++) {
    // jedno od 4 susedna polja
    int vv = v + dv[i], kk = k + dk[i];
    // ako je ono dostupno
    if (0 <= vv && vv < bv && 0 <= kk && kk < bk &&
        !A[vv][kk] && !posecen[vv][kk]) {
      // racunamo duzinu puta od susednog polja do cilja
      int duzina = najduziProhodanPut(A, bv, bk, posecen, vv, kk, vCilj, kCilj);
      // ako put postoji
      if (duzina >= 0)
        // azuriramo duzinu najduzeg puta iz tekuceg polja
        maksDuzina = max(maksDuzina, duzina+1);
    }
  }
  // zavrsena je analiza tekuceg polja i njega nadalje tumacimo slobodnim
```

```
  posecen[v][k] = false;

  return maksDuzina;
}

// ulazna tacka za korisnika
int najduziProhodanPut(const vector<vector<bool>>& A, int bv, int bk,
                       int v, int k, int vCilj, int kCilj) {
  vector<vector<bool>> posecen(bv, vector<bool>(bk, false));
  return najduziProhodanPut(A, bv, bk, posecen, v, k, vCilj, kCilj);
}

int main() {
  int bv, bk;
  vector<vector<bool>> A;
  citajMatricu(A, bv, bk);
  int vStart, kStart, vCilj, kCilj;
  cin >> vStart >> kStart >> vCilj >> kCilj;
  cout << najduziProhodanPut(A, bv, bk, vStart, kStart, vCilj, kCilj) << endl;
  return 0;
}
```

Задатак: Најдужи пут низбрдо

У граду постоји m улица у правцу исток-запад и n улица у правцу север-југ. Свака улица исток-запад се сече са сваком улицом север-југ и на тај начин се формира $m \cdot n$ раскрсница. Матрицом A_{mxn} дате су надморске висине раскрсница. Пера вози бицикл улицама града и жели да стигне од раскрснице (v_s, k_c) до раскрснице (v_c, k_c) али тако да иде само низбрдо, прецизније увек на путу се креће са раскрснице веће надморске висине на раскрсницу мање надморске висине. Написати програм којим се одређује дужина најдужег Периног пута, дужина пута је број раскрсница на путу (укључујући и први и последњу раскрсницу).

Улаз: Прва линија стандардног улаза садржи димензије матрице m и n ($1 < m, n \leq 100$), два природна броја одвојена једним размаком. Затим се на стандардном улазу налазе елементи матрице (у m линија по n бројева одвојених са по једним размаком). У последњој линији стандардног улаза налазе се 4 природна броја v_s, k_s, v_c, k_c ($0 \leq v_s, v_c < m, 0 \leq k_s, k_c < n$).

Излаз: На стандардном излазу приказати природан број који представља дужину најдужег пута низбрдо од позиције (v_s, k_s) до дате позиције (v_c, k_c). Ако пут низбрдо не постоји приказати поруку `ne postoji`.

Пример

Улаз	*Излаз*
4 5 12 10 11 9 5 10 8 5 3 6 4 7 4 2 10 2 6 7 8 5 1 0 3 0	5

Решење

Основна техника за решавање овог проблема је рекурзивна исцрпна претрага. Дефинисаћемо функцију која прима координате текуће раскрснице и израчунава најдужу вожњу низбрдо тј. највећи број раскрсница које се пређу од текуће раскрснице до циља (функција ће вратити 0 ако таква вожња није могућа). Ако је текућа раскрсница једнака циљној, већ смо стигли и укупно обилазимо једну раскрсницу (то је она на којој се налазимо). У супротном анализирамо четири суседне раскрснице датој (водећи рачуна да не испаднемо из опсега матрице) и из сваке од њих која је на нижој надморској висини од текуће рекурзивно покрећемо претрагу. На крају одређујемо и враћамо максимум путева који су нас довели до циља (то су они за које рекурзивни позив враћа позитивну вредност). Ако такав пут не постоји, враћамо нулу.

Нагласимо да се једном када се померимо са текуће раскрснице више не можемо вратити на њу путевима који иду само низбрдо. Стога није неопходно памтити раскрснице које смо већ обишли.

```cpp
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> visina;
int bv, bk;
int vCilj, kCilj;

int putNizbrdo(int v, int k) {
  if (v == vCilj && k == kCilj)
    return 1;

  int maksPut = 0;
  int dv[] = {1, -1, 0, 0};
  int dk[] = {0, 0, 1, -1};
  for (int i = 0; i < 4; i++) {
    int vv = v + dv[i], kk = k + dk[i];
    if (0 <= vv && vv < bv && 0 <= kk && kk < bk &&
        visina[vv][kk] < visina[v][k]) {
      int put = putNizbrdo(vv, kk);
      if (put > 0)
        maksPut = max(maksPut,  put + 1);
    }
  }
  return maksPut;
}

int main() {
  ios_base::sync_with_stdio(false);
  cin >> bv >> bk;
  visina.resize(bv, vector<int>(bk));
  for (int i = 0; i < bv; i++)
    for (int j = 0; j < bk; j++)
      cin >> visina[i][j];
  int vStart, kStart;
  cin >> vStart >> kStart >> vCilj >> kCilj;
  int maksPut = putNizbrdo(vStart, kStart);
  if (maksPut == 0)
    cout << "ne postoji" << endl;
  else
    cout << maksPut << endl;
  return 0;
}
```

Пошто се током рекурзивне претраге више пута понављају потпуно идентични рекурзивни позиви, решење се може знатно убрзати динамичким програмирање (на пример, техником мемозиације). За мемоизацију користимо матрицу у коју уписујемо резултат функције за свако поље на коме је позвана.

```cpp
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> visina;
int bv, bk;
int vCilj, kCilj;
vector<vector<int>> memo;
```

```
int putNizbrdo(int v, int k) {
  if (v == vCilj && k == kCilj)
    return 1;

  if (memo[v][k] != -1)
    return memo[v][k];

  int maksPut = 0;
  int dv[] = {1, -1, 0, 0};
  int dk[] = {0, 0, 1, -1};
  for (int i = 0; i < 4; i++) {
    int vv = v + dv[i], kk = k + dk[i];
    if (0 <= vv && vv < bv && 0 <= kk && kk < bk &&
        visina[vv][kk] < visina[v][k]) {
      int put = putNizbrdo(vv, kk);
      if (put > 0)
        maksPut = max(maksPut,  put + 1);
    }
  }
  return memo[v][k] = maksPut;
}

int main() {
  ios_base::sync_with_stdio(false);
  cin >> bv >> bk;
  visina.resize(bv, vector<int>(bk));
  for (int i = 0; i < bv; i++)
    for (int j = 0; j < bk; j++)
      cin >> visina[i][j];
  int vStart, kStart;
  cin >> vStart >> kStart >> vCilj >> kCilj;

  memo.resize(bv, vector<int>(bk, -1));

  int maksPut = putNizbrdo(vStart, kStart);
  if (maksPut == 0)
    cout << "ne postoji" << endl;
  else
    cout << maksPut << endl;
  return 0;
}
```

Задатак: Уклањање погрешних заграда

Ниска поред осталих карактера садржи и мале заграде ((и)), али могуће је да оне нису исправно упарене. Потребно је обрисати што мање карактера ниске да би се добила ниска у којој су заграде исправно упарене. Напиши програм који исписује све могуће ниске у којима су заграде исправно упарене а које су добијене брисањем што мањег броја карактера.

Улаз: Са стандардног улаза се учитава ниска дужине највише 50 карактера.

Излаз: На стандардни излаз исписати све тражене ниске у лексикографском редоследу.

Пример 1

Улаз	*Излаз*
`()())()`	`(())()` `()()()`

Пример 2

Улаз	*Излаз*
`(abc()+def)))(`	`(abc()+def)` `(abc(+def))`

Решење

Задатак можемо решити претрагом у дубину. Претпоставићемо да су полазна ниска и ниске које се од ње добијају брисањем заграда (друге карактере нема смисла брисати, јер не утичу на упареност заграда) стања кроз која се пролази. Полазно стање је оно које одговара полазној ниски, а завршна стања су сва она која одговарају нискама са исправно упареним заградама. Тада се задатак решава слично проналажењу најкраћег пута кроз лавиринт (види задатак Пут кроз лавиринт).

Дефинисаћемо помоћну функцију која проверава да ли су у датој листи заграде исправно упарене (довољно је одржавати бројач отворених заграда, увећавати га при наиласку на отворену заграду, умањивати га при наиласку на затворену заграду и водити рачуна да никада не постане негативан а да на крају дође на нулу).

Стања (ниске) стављамо у ред, кренувши од почетне ниске. Након тога узимамо ниске из реда, све док се ред не испразни. За сваку узету ниску проверавамо да ли су у њој заграде исправно упарене. Ако јесу, ниску смештамо у скуп ниски које треба на крају исписати у лексикографски сортираном редоследу (за то можемо користити библиотечку колекција за представљање сортираних скупова). У супротном пролазимо кроз ниску, уклањамо једну по једну заграду и све тако добијене ниске смештамо у ред.

Једна могућа оптимизација се може направити ако се примети да се у ред често убацују потпуно идентичне ниске, за чим очигледно нема потребе. Ако се нека ниска добијена избацивањем неке заграде већ налази у реду (што можемо утврдити одржавањем скупа ниски које се налазе у реду), нема је потребе додатно убацивати у ред.

Иако је ово решење могуће додатно оптимизовати, с обзиром на релативно малу димензију улаза, за тим нема потребе.

Задатак: Братска подела

Дат је скуп од n предмета, за сваки предмет позната је његова вредност (реалан број). Предмете треба да поделе два брата тако да се укупна вредност предмета које појединачно браћа добијају минимално разликују. При подели предмета сваки брат добија целе предмете и сваки предмет после поделе припада неком брату. Написати програм којим се одређује минимална разлика вредности коју браћа добијају при братској подели.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 10$). Следећих n линија садрже реалне бројеве, сваки у посебној линији, који представљају вредности предмета.

Излаз: На стандардном излазу приказати у једној линији минималну разлику вредности добијену при братској подели, разлику приказати на две децимале.

Пример

Улаз	*Излаз*

`4`
`3.5`
`1.7`
`8.0`
`1.2`
	`1.60`

Решење

Задатак се грубом силом решава тако што се наброје сви подскупови скупа од n елемената. Предмети који припадају подскупу се тумаче као предмети првог брата, док се предмети који не припадају том подскупу тумаче као предмети другог брата. Набрајање свих подскупова могуће је вршити тако што се за дати подскуп пронађе наредни подскуп у лексикографском редоследу (слично као у задатку Следећа варијација или Сви подскупови лексикографски). Мана овог решења је то што се за сваки подскуп изнова рачунају збирови предмета једног и другог брата.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <limits>
#include <cmath>

using namespace std;

// pronalazi sledeci podskup u leksikografskom redosledu (skup je
```

```
// predstavljen nizom logickih vrednosti) i vraca true akko takav
// skup postoji
bool sledeciPodskup(vector<bool>& skup) {
  int n = skup.size();
  int i = n-1;
  while (i >= 0 && skup[i])
    skup[i--] = false;
  if (i < 0)
    return false;
  skup[i] = true;
  return true;
}

// odredjuje razliku vrednosti predmeta dva brata, ako su poznate
// vrednosti svih predmeta i skup predmeta koji pripadaju prvom bratu
double razlika(const vector<double>& predmeti,
               const vector<bool>& predmetiPrvog) {
  int n = predmeti.size();
  double prvi = 0.0, drugi = 0.0;
  for (int i = 0; i < n; i++)
    if (predmetiPrvog[i])
      prvi += predmeti[i];
    else
      drugi += predmeti[i];
  return abs(prvi - drugi);
}

// funkcija vraca minimalnu razliku vrednosti koja se moze dobiti
// izmedju dva brata, ako oni medjusobno treba da podele sve predmete
// cije su vrednosti date
double podela(const vector<double>& predmeti) {
  int n = predmeti.size();
  vector<bool> predmetiPrvog(n, false);
  double minRazlika = numeric_limits<double>::max();
  do {
    double r = razlika(predmeti, predmetiPrvog);
    if (r < minRazlika)
      minRazlika = r;
  } while (sledeciPodskup(predmetiPrvog));
  return minRazlika;
}

int main() {
  // ucitavamo vrednosti predmeta
  int n;
  cin >> n;
  vector<double> predmeti(n);
  for (int i = 0; i < n; i++)
    cin >> predmeti[i];

  // odredjujemo i ispisujemo najmanju razliku
  cout << fixed << setprecision(2) << showpoint
       << podela(predmeti) << endl;

  return 0;
}
```

Испитивање свих могућности можемо реализовати и рекурзивном исцрпном претрагом. За сваки предмет испитујемо могућност да припадне једном и да припадне другом брату. Рекурзивна функција прима низ од n

предмета које је преостало да распореди, збир вредности већ распоређених предмета који су припали једном и збир вредности већ распоређених предмета који су припали другом брату. Излаз из рекурзије је случај када нема више предмета које треба распоредити и у том случају знамо разлику вредности предмета које су браћа добила. У супротном први (или последњи) предмет из низа додајемо првом брату и рекурзивно делимо преостале предмете, затим исти предмет додајемо другом брату и рекурзивно делимо преостале предмете и као резултат враћамо мању од разлика које су постигнуте на ова два начина.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <iomanip>

using namespace std;

// preostalih n predmeta treba podeliti izmedju dva brata koji na osnovu
// podele predmeta za sada imaju zbirPrvog i zbirDrugog vrednosti
// funkcija vraca minimalnu razliku vrednosti predmeta izmedju dva brata
double podela(const vector<double>& predmeti, int n,
              double zbirPrvog, double zbirDrugog) {
  // nema vise predmeta pa znamo minimalnu razliku
  if (n == 0)
    return abs(zbirPrvog - zbirDrugog);

  // analiziramo mogucnost da poslednji predmet damo jednom i
  // mogucnost da poslednji predmet damo drugom bratu
  return min(podela(predmeti, n-1, zbirPrvog + predmeti[n-1], zbirDrugog),
             podela(predmeti, n-1, zbirPrvog, zbirDrugog + predmeti[n-1]));
}


// funkcija vraca minimalnu razliku vrednosti koja se moze dobiti
// izmedju dva brata, ako oni medjusobno treba da podele sve predmete
// cije su vrednosti date
double podela(const vector<double>& predmeti) {
  // delimo svih n predmeta pri cemu braca u startu nemaju nista
  int n = predmeti.size();
  return podela(predmeti, n, 0.0, 0.0);
}

int main() {
  // ucitavamo vrednosti predmeta
  int n;
  cin >> n;
  vector<double> predmeti(n);
  for (int i = 0; i < n; i++)
    cin >> predmeti[i];

  // odredjujemo i ispisujemo najmanju razliku
  cout << fixed << setprecision(2) << showpoint
       << podela(predmeti) << endl;

  return 0;
}
```

Могуће је извршити и одређена одсецања у исцрпној претрази. Наиме, ако знамо збир преосталих предмета које треба распоредити и ако утврдимо да би доделом свих тих предмета брату који је тренутно добио мање тај брат и даље имао мању вредност, онда је јасно да се разлика минимизује баш на тај начин. На пример, ако је први брат добио 100, други 10, и треба распоредити још предмете од 10, 15 и 20, јасно је да све њих

треба дати другом брату и да се тиме добија разлика од 45. Овим се избегава анализа 7 могућности (да се неки од тих предмета додели првом брату). Дакле, пожељно је да прво поделимо велике предмете, а онда да ове мање предмете делимо овако групно. Стога низ вредности на почетку сортирамо тако да се у старту врши подела предмета са већом вредношћу.

Још једна оптимизација је да се разлика не мења ако браћа размене све предмете које имају. Да би се ова симетрија разбила (и избегла анализа суштински идентичних подела), у старту ћемо одабрати да се први предмет додели првом брату.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <functional>
#include <iomanip>

using namespace std;

double podela(const vector<double>& predmeti, int n,
              double zbirPredmeta,
              double zbirPrvog, double zbirDrugog) {
  // nema vise predmeta
  if (n == 0)
    return abs(zbirDrugog - zbirPrvog);

  // razlika u korist drugog je prevelika i isplati se sve predmete
  // dati prvom bratu
  if (zbirPrvog + zbirPredmeta < zbirDrugog)
    return zbirDrugog - (zbirPrvog + zbirPredmeta);

  // razlika u korist prvog je prevelika i isplati se sve predmete
  // dati drugom bratu
  if (zbirDrugog + zbirPredmeta < zbirPrvog)
    return zbirPrvog - (zbirDrugog + zbirPredmeta);

  // analiziramo mogucnost da poslednji predmet damo jednom i
  // mogucnost da poslednji predmet damo drugom bratu
  return min(podela(predmeti, n-1, zbirPredmeta - predmeti[n-1],
                    zbirPrvog + predmeti[n-1], zbirDrugog),
             podela(predmeti, n-1, zbirPredmeta - predmeti[n-1],
                    zbirPrvog, zbirDrugog + predmeti[n-1]));
}

// funkcija vraca minimalnu razliku vrednosti koja se moze dobiti
// izmedju dva brata, ako oni medjusobno treba da podele sve predmete
// cije su vrednosti date
double podela(vector<double>& predmeti) {
  // sortiramo predmete (da bi odsecanje bilo efikasnije)
  sort(begin(predmeti), end(predmeti));

  // izracunavamo ukupan zbir svih predmeta
  int n = predmeti.size();
  double ukupanZbir = 0.0;
  for (int i = 0; i < n; i++)
    ukupanZbir += predmeti[i];

  // delimo svih n predmeta pri cemu prvi predmet u startu dajemo
  // prvom bratu (da bi se razbila simetrija)
  return podela(predmeti, n-1, ukupanZbir - predmeti[n-1], predmeti[n-1], 0.0);
```

```
}

int main() {
  // ucitavamo vrednosti predmeta
  int n;
  cin >> n;
  vector<double> predmeti(n);
  for (int i = 0; i < n; i++)
    cin >> predmeti[i];

  // odredjujemo i ispisujemo najmanju razliku
  cout << fixed << setprecision(2) << showpoint
       << podela(predmeti) << endl;

  return 0;
}
```

Задатак: Збир суседних пун квадрат

Низ `8 1 15 10 6 3 13 12 4 5 11 14 2 7 9` је карактеристичан по томе што представља пермутацију бројева од 1 до 15 и збир било која два суседна елемента је квадрат неког природног броја. Напиши програм који за дато n одређује лексикографски најмању пермутацију бројева од 1 до n у којој је збир било која два суседна елемента квадрат неког природног броја.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 45$).

Излаз: На стандардни излаз исписати тражену пермутацију (сви бројеви у истом реду праћени са по једним размаком) или текст `nema` ако тражена пермутација не постоји.

Пример 1

Улаз	*Излаз*
12	nema

Пример 2

Улаз	*Излаз*
17	16 9 7 2 14 11 5 4 12 13 3 6 10 15 1 8 17

Решење

Задатак решавамо исцрпном бектрекинг претрагом. Тражени низ градимо елемент по елемент. На прво место уписујемо редом један по један елемент од 1 до n, а затим позивамо рекурзивну функцију да попуни остатак низа. Та функција на текуће место у низу покушава да постави све оне елементе који са претходним елементом низа дају потпун квадрат (низ свих пуних квадрата можемо израчунати унапред), а који се не јављају у до тада попуњеном делу низа (посебно ћемо одржавати скуп свих до тада попуњених елемената у облику низа логичких вредности таквог да је на месту a вредност тачно ако и само ако се елемент a јавља у попуњеном делу низа). Успешан излаз из рекурзије је када је цео низ попуњен (када текућа позиција која се попуњава постане једнака дужини низа).

```
#include <iostream>
#include <vector>

using namespace std;

bool zbir_susednih_pun_kvadrat(vector<int>& niz, int m,
                               vector<bool>& iskoriscen, const vector<int>& kvadrati) {
  if (m == niz.size()) {
    for (int x : niz)
      cout << x << " ";
    cout << endl;
    return true;
  } else {
    for (int k : kvadrati) {
      int dopuna = k - niz[m-1];
      if (1 <= dopuna && dopuna <= niz.size() && !iskoriscen[dopuna]) {
        niz[m] = dopuna;
```

```
        iskoriscen[dopuna] = true;
        if (zbir_susednih_pun_kvadrat(niz, m+1, iskoriscen, kvadrati))
          return true;
        iskoriscen[dopuna] = false;
      }
    }
  }
  return false;
}

bool zbir_susednih_pun_kvadrat(int n) {
  vector<bool> iskoriscen(n+1, false);
  vector<int> niz(n);
  vector<int> kvadrati;
  for (int k = 1; k*k <= n+(n-1); k++)
    kvadrati.push_back(k*k);

  for (int i = 1; i <= n; i++) {
    niz[0] = i;
    iskoriscen[i] = true;
    if (zbir_susednih_pun_kvadrat(niz, 1, iskoriscen, kvadrati))
      return true;
    iskoriscen[i] = false;
  }
  return  false;
}

int main() {
  int n;
  cin >> n;

  if (!zbir_susednih_pun_kvadrat(n))
    cout << "nema" << endl;

  return 0;
}
```

Задатак: Подела на палиндромске подниске

Напиши програм који одређује све начине да се ниска подели на узастопне непразне подниске које су палиндроми (читају се исто са лева и са десна).

Улаз: Са стандардног улаза се учитава ниска која има највише 50 карактера енглеског алфабета.

Излаз: На стандардни излаз исписати све тражене поделе ниске на палиндромске подниске (подниске писати у истом реду, раздвојене са по једним размаком). Ако постоји више решења, исписивати прво она која почињу краћим поднискама.

Пример

Улаз	*Излаз*
sirimamiris	s i r i m a m i r i s s i r i m a m iri s s i r i mam i r i s s i r i mam iri s s i r imami r i s s i rimamir i s s iri m a m i r i s s iri m a m iri s s iri mam i r i s s iri mam iri s s irimamiri s sirimamiris

Решење

С обзиром на релативно малу дужину улазне ниске, задатак можемо решити рекурзивном исцрпном претрагом. Главна функција врши рекурзивну претрагу и прима низ палиндрома који је добијен разлагањем неког почетног дела ниске, као и преостали део ниске који треба разложити на палиндромске подниске. Иницијално је тај низ празан и потребно је разложити целу ниску. Ако је преостали део ниске празан, тада је цела ниска разложена и исписујемо палиндроме смештене у низ. У супротном, анализирамо све префиксе преосталог дела (кренувши од једночланог) и сваки од њих који је палиндром (дефинисаћемо помоћну функцију која проверава да ли је подниска одређена позицијама првог и последњег слова палиндром, слично као у задатку Ниска палиндром), уклањамо га из ниске, премештамо га у низ и настављамо рекурзивно разлагање дела ниске иза тог префикса.

```
#include <iostream>
#include <vector>

using namespace std;

bool jePalindrom(const string& s, int Od, int Do) {
  for (int i = Od, j = Do; i < j; i++, j--)
    if (s[i] != s[j])
      return false;
  return true;
}

void podelaNaPalindrome(const string& s, int i, vector<string>& palindromi) {
  if (i == s.length()) {
    for (const string& s : palindromi)
      cout << s << " ";
    cout << endl;
  } else {
    for (int j = i; j < s.length(); j++)
      if (jePalindrom(s, i, j)) {
        palindromi.push_back(s.substr(i, j-i+1));
        podelaNaPalindrome(s, j+1, palindromi);
        palindromi.pop_back();
      }
  }
}

void podelaNaPalindrome(const string& s) {
  vector<string> palindromi;
  podelaNaPalindrome(s, 0, palindromi);
}

int main() {
```

```
  string s;
  cin >> s;
  podelaNaPalindrome(s);
  return 0;
}
```

Глава 6

Подели па владај

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање обједињавањем (MergeSort)

Алгоритам сортирања обједињавањем дели низ на два дела чије се дужине разликују највише за 1 (уколико је дужина низа паран број, онда су ова два дела једнаких дужина), рекурзивно сортира сваки од њих и затим обједињује сортиране половине. За обједињавање је неопходно користити додатни, помоћни низ, а на крају се обједињени низ копира у полазни низ. Излаз из рекурзије је случај једночланог низа (случај празног низа не може да наступи осим ако је полазни низ празан).

Кључна операција у овом алгоритму је операција обједињавања сортираних низова, техником два показивача. Њена имплементација описана је у задатку Обједињавање. На пример, обједињавањем сортираних низова `a` и `b` добија се сортирани низ `c`. Два већ сортирана низа могу се објединити у трећи сортирани низ само једним проласком кроз низове (тј. у линеарном времену $O(m+n)$ где су m и n димензије полазних низова).

```
a:  1 3 4 7 9 11                  b:  2 5 8 9 10 12 14
          c: 1 2 3 4 5 7 8 9 9 10 11 12 14
```

Функција сортирања обједињавањем сортира део низа $a[l, d]$, уз коришћење низа tmp као помоћног. Променљива n чува број елемената који се сортирају у оквиру овог рекурзивног позива, а променљива s чува средишњи индекс у низу између l и d. Рекурзивно се сортира $n_1 = \lfloor \frac{n}{2} \rfloor$ елемената између позиција l и $s-1$ и $n_2 = n - \lfloor \frac{n}{2} \rfloor$ елемената између позиција s и d. Након тога, сортирани поднизови обједињују се у помоћни низ. Пошто се више не обједињују цели низови, већ делови једног низа, функцију обједињавања морамо мало прилагодити.

Помоћни низ може се пре почетка сортирања алоцирати и користити кроз рекурзивне позиве.

Добијена функција сортирања има гарантовану сложеност најгорег случаја $O(n \log n)$, што значи да је много бржа од функција заснованих на сортирању селекцијом или сортирању уметањем чија је сложеност $O(n^2)$.

```
// ucesljava deo niza a iz intervala pozicija [i, m] i deo niza b iz
// intervala pozicija [j, n] koji su vec sortirani tako da se dobije
// sortiran rezultat koji se smesta u niz c, krenuvsi od pozicije k
void merge(vector<int>& a, int i, int m,
           vector<int>& b, int j, int n,
           vector<int>& c, int k) {
  while (i <= m && j <= n)
    c[k++] = a[i] <= b[j] ? a[i++] : b[j++];
  while (i <= m)
    c[k++] = a[i++];
```

```
  while (j <= n)
    c[k++] = b[j++];
}

// sortira deo niza a iz intervala pozicija [l, d] koristeci
// niz tmp kao pomocni
void merge_sort(vector<int>& a, int l, int d, vector<int>& tmp) {
  // ako je segment [l, d] jednoclan ili prazan, niz je vec sortiran
  if (l < d) {
    // sredina segmenta [l, d]
    int s = l + (d - l) / 2;
    // sortiramo segment [l, s]
    merge_sort(a, l, s, tmp);
    // sortiramo segment [s+1, d]
    merge_sort(a, s+1, d, tmp);

    // ucesljavamo segmente [l, s] i [s+1, d] smestajuci rezultat u
    // niz tmp
    merge(a, l, s, a, s+1, d, tmp, l);
    // vracamo rezultat iz niza tmp nazad u niz a
    for (int i = l; i <= d; i++)
      a[i] = tmp[i];

    // moze i pomocu biblioteckih funkcija
    /*
    merge(next(a.begin(), l), next(a.begin(), s+1),
          next(a.begin(), s+1), next(a.begin(), d+1),
          next(tmp.begin(), l));
    copy(next(tmp.begin(), l), next(tmp.begin(), d+1), next(a.begin(), l));
    */
  }
}

// sortira niz a
void merge_sort(vector<int>& a) {
  // alociramo pomocni niz
  vector<int> tmp(a.size());
  // pozivamo funkciju sortiranja
  merge_sort(a, 0, a.size() - 1, tmp);
}
```

Брзо сортирање (QuickSort)

У сваком кораку алгоритма сортирања један елемент (обично називан *пивот*) се доводи на своје место (пожељно близу средине низа). Да би након тога, проблем могао бити сведен на сортирање два мања подниза, потребно је приликом довођења пивота на своје место груписати све елементе мање или једнаке од њега лево од њега, а све елементе веће од њега десно од њега (ако се низ сортира неопадајуће). То прегруписавање елемената низа, *корак партиционисања* кључни је корак алгоритма брзог сотирања.

Брзо сортирање се може имплементирати на следећи начин. Позив `qsort_(a, l, d)` sortira deo niza $a[l, d]$. Партиционисање се врши техником два показивача. Слична техника је приказана у задацима Двобојка и Тробојка.

Након партиционисање рекурзивно се сортирају лева и десна половина низа. Излаз из рекурзије представља случај када је низ (тј. његов део $a[l, d]$) празан или једночлан (такав низ је већ сортиран).

Сложеност најгорег случаја овог алгоритма може бити квадратна тј. $O(n^2)$, ако се стално дешава да пивот дели низ на две неравномерне целине. Ипак, може се доказати да је просечна сложеност овог алгоритма $O(n \log n)$ и у пракси он показује веома добре резултате (за разлику од сортирања обједињавањем не троши се време на померање елемената између помоћног и главног низа).

```
// soritra segment pozicija [l, d] u nizu a
void quick_sort(vector<int>& a, int l, int d) {
  // ako segment [l, d] jedan ili nula elementa on je vec sortiran
  if (l < d) {
    // za pivot uzimamo proizvoljan element segmenta
    swap(a[l], a[l + rand() % (d - l + 1)]);
    // particionisemo niz tako da se u njemu prvo javljaju elementi
    // manji ili jednaki pivotu, a zatim veci od pivota
    // tokom rada vazi [l, k] su manji ili jednaki pivotu
    // (k, i) su veci od pivota, [i, d] su jos neispitani
    int k = l;
    for (int i = l+1; i <= d; i++)
      if (a[i] <= a[l])
        swap(a[i], a[++k]);
    // razmenjujemo pivot sa poslednjim manjim ili jednakim elementom
    swap(a[l], a[k]);
    // rekurzivno sortiramo deo niza levo i desno od pivota
    quick_sort(a, l, k - 1);
    quick_sort(a, k + 1, d);
  }
}

// sortira niz a
void quick_sort(vector<int>& a) {
  // poziv pomocne funkcije koja u nizu a sortira segment pozicija [0, n-1]
  quick_sort(a, 0, a.size() - 1);
}
```

Види другачија решења овог задатка.

Задатак: Медијана

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Библиотечка функција

У библиотеци језика C++ налази се функција `nth_element` која се може употребити да ефикасно израчуна елемент низа који би се након сортирања нашао на датој позицији, гарантујући при том да ће сви елементи лево од њега бити мањи или једнаки свим елементима десно од њега. Функцији се прослеђују три итератора (на почетак, непосредно иза краја и на позицију на којој се тражи елемент). Уз помоћ ове функције медијану можемо пронаћи веома једноставно (при чему, ако је број елемената низа паран, функцију `nth_element` морамо позвати два пута).

Сложеност функције `nth_element` је $O(n)$, где је n број елемената низа.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>

using namespace std;

unsigned nti_element(vector<unsigned>& a, int n) {
  auto it = next(a.begin(), n);
  nth_element(a.begin(), it, a.end());
  return *it;
```

```
}

double medijana(vector<unsigned>& a) {
  if (a.size() % 2 != 0)
    return nti_element(a, a.size() / 2);
  else
    return ((double)nti_element(a, a.size() / 2 - 1) +
            (double)nti_element(a, a.size() / 2)) / 2.0;
}

int main() {
  int n;
  cin >> n;
  unsigned c0, c1;
  cin >> c0 >> c1;
  vector<unsigned> a(n);
  cin >> a[0];
  for (int i = 1; i < n; i++)
    a[i] = c0 * a[i-1] + c1;
  /*
  sort(begin(a), end(a));
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;
  */
  cout << fixed << showpoint << setprecision(2) << medijana(a) << endl;
  return 0;
}
```

QuickSelect

Ако на располагању немамо библиотечке функције, а желимо решење које је ефикасније од сортирања можемо употребити модификацију алгоритма QuickSort која је позната под именом QuickSelect. Покажимо варијанту која одређује елемент низа на позицији n (која се налази у интервалу $[l, d]$, где су l и d границе дела низа који се обрађује.

Алгоритам почиње одабиром пивотирајућег елемента и партиционисањем низа (види задатке Двобојка и Тробојка, као и имплементацију алгоритма QuickSort приказану у задатку Сортирање бројева). Након партиционисања постоје три могућности. Нека је позиција пивота након партиционисања p. Једна је да се пивот налази баш на траженој позицији n, тј. да је $p = n$ - у том случају функција враћа пивот и завршава са радом. Друга могућност је да је $n < p$ и тада се алгоритам примењује на део низа $[l, p-1]$. На крају, трећа могућност је да је $n > p$ и тада се алгоритам примењује на део низа $[p+1, d]$. Основна имплементација може бити рекурзивна (по узору на QuickSort функција прима параметре l и d), међутим, пошто се у варијанти QuickSelect врши само један рекурзивни позив и то као репни, рекурзију је веома једноставно елиминисати.

Као и у случају алгоритма QuickSort и сложеност алгоритма QuickSelect зависи од тога колико среће имамо да пивот равномерно подели интервал $[l, d]$. Ако би се у сваком кораку десило да пивот упадне на средину интервала, укупан број поређења и размена био би $O(n)$ (заиста, у првом кораку то се ради n пута, у другом $n/2$, у трећем $n/4$ и тако даље, а збир тих елемената се одозго може ограничити са $2n$). Са друге стране, ако би пивот стално био близак неком од два краја интервала $[l, d]$, тада би сложеност алгоритма била $O(n^2)$ (заиста, у првом кораку бисмо имали n поређења, у другом $n-1$, у трећем $n-2$ што у збиру даје $n(n+1)/2$).

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
#include <random>

using namespace std;

int random_value(int i, int j) {
```

```
  mt19937_64 random_generator;
  uniform_int_distribution<int> dist(i, j);
  return dist(random_generator);
}

unsigned ntiElement(vector<unsigned>& a, int l, int d, int n) {
  while (true) {
    swap(a[l], a[random_value(l, d)]);
    int i = l + 1, j = d;
    while (i <= j) {
      if (a[i] < a[l])
        i++;
      else if (a[j] > a[l])
        j--;
      else
        swap(a[i++], a[j--]);
    }
    swap(a[l], a[j]);
    if (n < j)
      d = j - 1;
    else if (n > j)
      l = j + 1;
    else
      return a[n];
  }
}

unsigned ntiElement(vector<unsigned>& a, int n) {
  return ntiElement(a, 0, a.size() - 1, n);
}


double medijana(vector<unsigned>& a) {
  if (a.size() % 2 != 0)
    return ntiElement(a, a.size() / 2);
  else
    return ((double)ntiElement(a, a.size() / 2 - 1) +
            (double)ntiElement(a, a.size() / 2)) / 2.0;
}

int main() {
  int n;
  cin >> n;
  unsigned c0, c1;
  cin >> c0 >> c1;
  vector<unsigned> a(n);
  cin >> a[0];
  for (int i = 1; i < n; i++)
    a[i] = c0 * a[i-1] + c1;
  cout << fixed << showpoint << setprecision(2) << medijana(a) << endl;
  return 0;
}
```

Нагласимо да избор алгоритма партиционисања може утицати на ефикасност, нарочито у случају када у низу има доста поновљених елемената. Алгоритам који партиционисање врши тако што низ обилази са два краја и врши размене је у том светлу доста ефикаснији од алгоритма који низ обилази само са једног краја (јер се у случају поновљених елемената показивачи сусрећу близу средине).

```
#include <iostream>
```

```cpp
#include <iomanip>
#include <vector>
#include <algorithm>
#include <random>

using namespace std;

int random_value(int i, int j) {
  mt19937_64 random_generator;
  uniform_int_distribution<int> dist(i, j);
  return dist(random_generator);
}

unsigned ntiElement(vector<unsigned>& a, int l, int d, int n) {
  swap(a[l], a[random_value(l, d)]);
  int p = l;
  for (int i = l + 1; i <= d; i++)
    if (a[i] < a[l])
      swap(a[i], a[++p]);
  swap(a[l], a[p]);
  if (n < p)
    return ntiElement(a, l, p - 1, n);
  else if (n > p)
    return ntiElement(a, p + 1, d, n);
  else
    return a[n];
}

unsigned ntiElement(vector<unsigned>& a, int n) {
  return ntiElement(a, 0, a.size() - 1, n);
}


double medijana(vector<unsigned>& a) {
  if (a.size() % 2 != 0)
    return ntiElement(a, a.size() / 2);
  else
    return ((double)ntiElement(a, a.size() / 2 - 1) +
            (double)ntiElement(a, a.size() / 2)) / 2.0;
}

int main() {
  int n;
  cin >> n;
  unsigned c0, c1;
  cin >> c0 >> c1;
  vector<unsigned> a(n);
  cin >> a[0];
  for (int i = 1; i < n; i++)
    a[i] = c0 * a[i-1] + c1;

  cout << fixed << showpoint << setprecision(2) << medijana(a) << endl;
  return 0;
}
```

Задатак: Збир k најбољих

Ученик је радио n задатака и за сваки задатак је добио одређени број поена. Одредити збир поена на k задатака које је најбоље урадио.

Улаз: У првој линији стандардног улаза унети природан број n ($1 \leq n \leq 10^6$) - број задатака које је ученик радио, у другој природан број k ($1 \leq k \leq n$) - број задатака које је најбоље урадио, а затим у следећих n линија број поена које је добио на задацима.

Излаз: Укупан број поена које је освојио на k најбоље оцењених задатака.

Пример

Улаз	*Излаз*
10 3 15 80 25 60 10 20 50 45 40 30	190

Решење

Сортирање целог низа

Задатак се може решити и тако што ће се низ сортирати библиотечком функцијом за сортирање (слично као у задатку Сортирање бројева). Ако се низ сортира нерастуће, онда је након сортирања потребно сабрати првих k елемената низа, а ако се сортира неопадајуће, онда је након сортирања потребно сабрати последњих k елемената низа (сортирање неопадајуће је обично подразумевани начин сортирања и лакше га је реализовати).

Ако се сортирање врши библиотечким функцијама, временска сложеност овог решења је $O(n \cdot \log(n))$, док је меморијска сложеност једнака $O(n)$.

Може се приметити да овакав алгоритам непотребно губи време прецизно одређујући редослед тј. сортирајући елементе који су мањи од првих k као и одређујући прецизан редослед првих k елемената (да би се сабрало првих k елемената они не морају бити поређани, већ је само потребно на почетак низа (или на крај) довести првих k елемената, а иза њих (или испред) поставити остале елементе тј. раздвојити те две групе елемената, при чему је редослед елемената унутар сваке од група ирелевантан.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>  // zbog greater<int>() u pozivu funkcije sort
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;

  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
```

```
  // sortiramo niz nerastuce
  sort(begin(a), end(a), greater<int>());

  // sabiramo prvih k elemenata niza
  int s = accumulate(begin(a), next(begin(a), k), 0);

  // ispisujemo rezultat
  cout << s << endl;
}
```

Модификовани алгоритам сортирања селекцијом

Једна могућа идеја која избегава сортирање елемената који су мањи од првих k је да се сортирање врши алгоритмом селекције (види задатак Сортирање бројева) који, подсетимо се, у сваком кораку на текуће место у низу доводи најмањи од преосталих елемената низа (у првом кораку се на прво место доводи највећи (или најмањи) елемент целог низа, у другом кораку се на друго место доводи највећи од свих елемената низа осим оног постављеног на прво место итд.). Приметимо да ће се након k корака на почетку низа наћи тачно k највећих елемената и ту се алгоритам може зауставити.

Пошто је за налажење најмањег елемента у низу потребно $O(n)$ операција, и то се ради k пута, временска сложеност овог алгоритма је $O(n \cdot k)$, док је меморијска сложеност $O(n)$.

```
#include <iostream>
#include <algorithm>
using namespace std;

const int N_MAX = 100000;
int a[N_MAX];

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // sortiramo niz primenjujuci k rundi algoritam sortiranja
  // selekcijom
  for (int i = 0; i < k; i++) {
    int minPoz = i;
    for (int j = i + 1; j < n; j++)
      if (a[j] > a[minPoz])
        minPoz = j;
    swap(a[i], a[minPoz]);
  }

  // izracunavamo i ispisujemo zbir elemenata niza
  int s = 0;
  for (int i = 0; i < k; i++)
    s += a[i];
  cout << s << endl;

  return 0;
}
```

QuickSelect

Алгоритам брзе селекције (QuickSelect) представља модификацију алгоритма брзог сортирања који се може употребити да се низ подели тако да на првих k највећих (или најмањих) елемената низа, а након тога

елементи мањи (или већи) од њих, при чему је редослед елемената у те две групе произвољан.

Алгоритам брзе селекције се може имплементирати и ручно и заснива на кораку партиционисања који у линеарној сложености елементе низа уређује тако да се прво у низу нађу елементи који су већи од неког датог елемента (тзв. пивота), затим пивот и након тога елементи који су мањи од пивота. Редослед елемената у свакој од ових група је потпуно произвољан. Ако се пивот јавља више пута, остала појављивања пивота могу бити било лево, било десно од пивота (често се узима да се лево од пивота налазе елементи већи или једнаки од њега, а десно елементи строго мањи од њега или обратно). За партиционисање се могу користити поступци описани у задацима Двобојка или Тробојка. Осовни алгоритам је рекурзиван и параметар рекурзије су границе l и d и број k, и задатак алгоритма је да део низа на позицијама $[l, d]$ реорганизује тако да на почетку буде k најмањих елемената тог дела низа. Излаз из рекурзије је када је интервал $[l, d]$ једночлан или празан (када је $l \geq d$) или када је $k \leq 0$. Након избора пивота и партиционисања претпоставимо да се пивот нашао на месту m. Ако је број елемената лево од пивота (вредност $m - l$) већа од k онда је довољно наћи k највећих елемената левог дела низа (јер су сви елементи у левом мањи од елемената у десном делу, јер их након партиционисања раздваја пивот) тј. извршити рекурзивни позив за интервал $[l, m - 1]$ и број k. У супротном се закључно са пивотом налази $m - l + 1$ од k највећих елемената низа и зато је потребно у десном делу одредити још $k - (m - l + 1)$ највећих елемената из тог дела, тако да се рекурзивни позив врши за интервал $[m + 1, d]$ и број $k - m + l - 1$. Приметимо да у функцији постоји само један рекурзивни позив и то репни, тако да се он може једноставно елиминисати.

Под претпоставком да ће пивот делити низ на делове који су отприлике једнаке величине, сложеност овог алгоритма је линеарна $O(n)$. Пошто се цео низ учитава и чува у меморији и просторна сложеност је $O(n)$.

```
#include <iostream>
#include <vector>
#include <utility>
using namespace std;

// QuickSelect - odredjujemo najvecih k elemenata niza a tj. niz permutujemo
// tako da se najvecih k elemenata nadju na prvih k pozicija (u proizvoljnom
// redosledu)
void qsortK(vector<int>& a, int l, int d, int k) {
  if (k <= 0 || l >= d)
    return;

  // niz particionisemo tako da se pivot (element a[l]) dovede na
  // svoje mesto, da ispred njega budu svi elementi koji su veci ili
  // jednaki od njega, a da iza njega budu svi elementi veci od njega
  int m = l;
  for (int t = l+1; t <= d; t++)
    if (a[t] >= a[l])
      swap(a[++m], a[t]);
  swap(a[m], a[l]);

  if (k < m - l)
    // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
    qsortK(a, l, m - 1, k);
  else
    // neki kod k najvecih su iza pivota - obradjujemo deo iza pivota
    qsortK(a, m+1, d, k - (m - l + 1));
}

// QuickSelect - pomocna funkcija zbog lepseg interfejsa
void qsortK(vector<int>& a, int k) {
  qsortK(a, 0, a.size() - 1, k);
}

int main() {
  ios_base::sync_with_stdio(false);
```

```
  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
      cin >> a[i];

  // odredjujemo prvih k najvecih elemenata niza
  qsortK(a, k);

  // sabiramo prvih k elemenata niza i ispisujemo rezultat
  int s = 0;
  for (int i = 0; i < k; i++)
    s += a[i];

  cout << s << endl;
  return 0;
}
```

Рачунање збира током партиционисања

Уместо да елементе прво распоредимо тако да k највећих елемената буде на почетку, па тек затим да их сабирамо, функција може бити дефинисана тако да истовремено распоређује елементе и рачуна њихов збир.

```
#include <iostream>
#include <vector>
using namespace std;

// QuickSelect - odredjujemo zbir k najvecih elemenata dela niza [l, d]
int zbirKNajvecih(vector<int>& a, int l, int d, int k) {
  if (k == 0)
    return 0;

  // niz particionisemo tako da se pivot (element a[l]) dovede na
  // svoje mesto, da ispred njega budu svi elementi koji su veci ili
  // jednaki od njega, a da iza njega budu svi elementi veci od njega
  int m = l;
  for (int t = l+1; t <= d; t++)
    if (a[t] >= a[l])
      swap(a[++m], a[t]);
  swap(a[m], a[l]);

  if (k < m - l + 1)
    // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
    return zbirKNajvecih(a, l, m - 1, k);
  else {
    int zbir = 0;
    for (int i = l; i <= m; i++)
      zbir += a[i];
    // neki kod k najvecih su iza pivota - obradjujemo deo iza pivota
    return zbir + zbirKNajvecih(a, m+1, d, k - (m - l + 1));
  }
}

// QuickSelect - pomocna funkcija zbog lepseg interfejsa
int zbirKNajvecih(vector<int>& a, int k) {
  return zbirKNajvecih(a, 0, a.size() - 1, k);
}
```

```
int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
      cin >> a[i];

  // odredjujemo zbir prvih k najvecih elemenata niza
  int zbir = zbirKNajvecih(a, k);

  cout << zbir << endl;
  return 0;
}
```

Пошто је рекурзија репна, она се лако елиминише.

```
#include <iostream>
#include <vector>
using namespace std;

// QuickSelect - zbir k najvecih elemenata niza
int zbirKNajvecih(vector<int>& a, int k) {
  int l = 0, d = a.size() - 1;
  int zbir = 0;
  while (k != 0) {
    // niz particionisemo tako da se pivot (element a[l]) dovede na
    // svoje mesto, da ispred njega budu svi elementi koji su veci ili
    // jednaki od njega, a da iza njega budu svi elementi veci od njega
    int m = l;
    for (int t = l+1; t <= d; t++)
      if (a[t] >= a[l])
        swap(a[++m], a[t]);
    swap(a[m], a[l]);

    if (k < m - l + 1)
      // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
      d = m - 1;
    else {
      for (int i = l; i <= m; i++)
        zbir += a[i];
      // neki kod k najvecih su iza pivota - obradjujemo deo iza pivota
      k -= m - l + 1;
      l = m + 1;
    }
  }
  return zbir;
}

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
```

```
    cin >> a[i];

  // odredjujemo zbir prvih k najvecih elemenata niza
  int zbir = zbirKNajvecih(a, k);

  cout << zbir << endl;
  return 0;
}
```

Библиотечка функција

У језику C++ библиотечка функција `nth_element` врши поделу низа тако да се на позицији n нађе елемент који ту и припада у сортираном редоследу, да се испред те позиције нађу елементи који су сви мањи или једнаки од њега, а да се иза те позиције нађу елементи који су сви већи или једнаки од њега. Функцији се прослеђује итератор на почетак дела низа (вектора) који се обрађује (обично добијен помоћу `begin`), итератор на неку позицију на средини низа и итератор који указује непосредно иза краја низа (обично добијен помоћу `end`). Ако средишњи итератор указује на n-ту позицију у низу након примене функције на тој позицији ће се наћи n-ти по величини елемент, док ће сви елементи лево од њега бити мањи или једнаки од свих елемената десно од њега. Рецимо и да постоји функција `partial_sort` која је слична претходној али уједно елементе испред дате позиције уређује (соритра) по величини, међутим, у овом случају то нам није потребно и тиме би се само непотребно губило време.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <numeric>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
      cin >> a[i];

  // niz particionisemo tako da je  n-ti element na svom mestu i da su
  // svi elementi ispred njega manji ili jednaki od svih elemenata iza
  nth_element(a.begin(), next(a.begin(), k), a.end(), greater<int>());

  // odredjujemo i ispisujemo zbir prvih k elemenata transformisanog niza
  cout << accumulate(a.begin(), next(a.begin(), k), 0) << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Силуета града

Са брода се виде зграде на обали велеграда. Дуж обале је постављена координатна оса и за сваку зграду се зна позиција левог краја, висина и позиција десног краја. Написати програм који израчунава силуету града.

Силуета је део-по-део константна функција и одређена је интервалима константности $(-\infty, x_0)$, $[x_0, x_1)$, $[x_1, x_2)$, ..., $[x_{n-1}, +\infty)$, одређеним тачкама поделе $x_0 < x_1 < \ldots < x_{n-1}$ и вредностима 0, h_0, ..., h_{n-2} и h функције на сваком од интервала.

$$\begin{array}{ccccccccccccc} & & 0 & & h_0 & & h_1 & & \dots & & h_{n-2} & & 0 & \\ -\infty & & x_0 & & x_1 & & x_2 & \dots & x_{n-2} & & x_{n-1} & & +\infty \end{array}$$

Подразумевамо да су крајње тачке $-\infty$ и $+\infty$ и да су вредности на тим интервалима једнаке нули. Дакле, део-по-део константна функција се може представити помоћу n тачака $x_0, \dots x_{n-1}$ и $n-1$ вредности $h_0, \dots, h_{n-2}$. Једноставности ради ми ћемо овакве функције представљати помоћу n уређених парова (x_0, h_0), (x_1, h_1), …, (x_{n-2}, h_{n-2}) и $(x_{n-1}, 0)$. Дакле, наш алгоритам прима низ уређених тројки који описује појединачне зграде, а враћа низ уређених парова који описује силуету.

Улаз: Са станадардног улаза се учитава број зграда n, а затим, у n наредних линија по три цела броја раздвојена са по једним размаком: леви крај зграде, десни крај зграде и висина зграде.

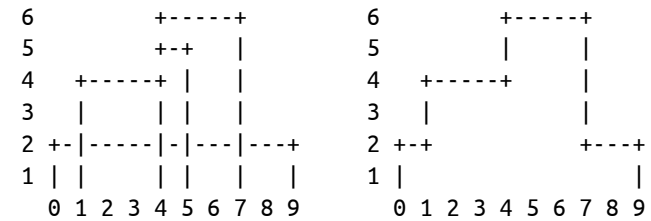
Излаз: На стандардни излаз исписати силуету описану преко низа промена висина.

Пример

Улаз	*Излаз*
4	0 2
1 4 4	1 4
4 5 5	4 6
4 7 6	7 2
0 9 2	9 0

Објашњење

Са леве стране су приказане зграде, а са десне силуета.



Решење

Подели па владај

Проблем можемо ефикасније решити техником подели-па-владај, веома слично алгоритму сортирања обједињавањем (енгл. merge sort). Кључна опаска је то да две силуете можемо објединити за исто време за које можемо објединити једну зграду у силуету. Пошто су резултујуће силуете сортиране можемо их обилазити уз одржавање два показивача и обједињавати веома слично обједињавању два сортирана низа бројева (као у задатку Обједињавање).

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
using namespace std;

// zgrada je odredjena pocetkom a, krajem b i visinom h
struct zgrada {
  int a, b, h;
  zgrada(int a = 0, int b = 0, int h = 0)
    : a(a), b(b), h(h) {
  }
};

// silueta je odredjena nizom promena
// svaka promena je odredjena koordinatom x i visinom h
struct promena {
  int x, h;
```

```
  promena(int x = 0, int h = 0)
    : x(x), h(h) {
  }
};


// integrisemo promenu (x, h) u postojecu siluetu
void dodajPromenu(vector<promena>& silueta, int x, int h) {
  if (silueta.size() > 0) {
    // poslednja promena tekuće siluete
    int xb = silueta.back().x, hb = silueta.back().h;
    // ako se promena dešava na istoj x koordinati kao prethodna
    // samo menjamo visinu
    if (x == xb)
      silueta.back().h = h;
    // ako nova promena ima istu visinu kao prethodna, preskačemo je
    else if (h != hb)
      // u suprotnom dodajemo novu promenu u niz
      silueta.emplace_back(x, h);
  } else
    // nema prethodne promene, pa novu promenu dodajemo na početak niza
    silueta.emplace_back(x, h);
}

// odredjuje se silueta zgrada na pozicijama [l, d]
vector<promena> silueta(const vector<zgrada>& zgrade, int l, int d) {
  vector<promena> rezultat;

  // silueta koja odgovara jednoj zgradi
  if (l == d) {
    rezultat.emplace_back(zgrade[l].a, zgrade[l].h);
    rezultat.emplace_back(zgrade[l].b, 0);
    return rezultat;
  }

  // određujemo posebno siluete za prvu i drugu polovinu zgrada
  int s = l + (d - l) / 2;
  vector<promena> rezultat_l = silueta(zgrade, l, s);
  vector<promena> rezultat_d = silueta(zgrade, s+1, d);

  // objedinjujemo dve siluete

  // tekući indeksi i visine u levoj i desnoj silueti
  int ll = 0, dd = 0;
  int Hl = 0, Hd = 0;
  // dok god postoji neka neobrađena promena
  while (ll < rezultat_l.size() || dd < rezultat_d.size()) {
    // određujemo novu tačku promene
    int x;
    // ako smo završili sa levom siluetom samo prebacujemo zgrade iz desne
    if (ll == rezultat_l.size()) {
      x = rezultat_d[dd].x; Hd = rezultat_d[dd].h;
      dd++;
    // ako smo završili sa desnom siluetom samo prebacujemo zgrade iz desne
    } else if (dd == rezultat_d.size()) {
      x = rezultat_l[ll].x; Hl = rezultat_l[ll].h;
      ll++;
    } else {
```

```
      // određujemo raniju od tekućih promena leve i desne siluete
      int xl = rezultat_l[ll].x;
      int xd = rezultat_d[dd].x;
      if (xl <= xd) {
        x = xl; Hl = rezultat_l[ll].h;
        ll++;
      } else {
        x = xd; Hd = rezultat_d[dd].h;
        dd++;
      }
    }

    // veća od dve tekuće visine
    int h = max(Hl, Hd);

    // integrišemo promenu (x, h) u tekuću rezultujući siluetu
    dodajPromenu(rezultat, x, h);
  }

  return rezultat;
}

// odredjuje se silueta niza zgrada
vector<promena> silueta(const vector<zgrada>& zgrade) {
   return silueta(zgrade, 0, zgrade.size() - 1);
}

int main() {
  // ucitavamo podatke o zgradama
  int n;
  cin >> n;
  vector<zgrada> zgrade(n);
  for (int i = 0; i < n; i++) {
    int a, b, h;
    cin >> a >> b >> h;
    zgrade[i] = zgrada(a, b, h);
  }

  // gradimo siluetu
  vector<promena> s = silueta(zgrade);

  // ispisujemo rezultat
  for (auto p : s)
    cout << p.x << " " << p.h << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Број инверзија

Напиши програм који одређује колико у низу има инверзија (позиција $0 \leq i < j < n$, таквих да је $a_i > a_j$.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$) и затим n целих бројева, сваки у посебном реду.

Излаз: На стандардни излаз исписати само тражени број инверзија.

Пример

Улаз	*Излаз*
5 3 1 4 2 5	3

Решење

Груба сила

Грубом силом се задатак решава тако што се помоћу угнежђених петљи испитају сви парови позиција $0 \leq i < j < n$ и преброје сви случајеви када је $a_i > a_j$ (бројимо елементе филтриране серије). Сложеност овог алгоритма одговара броју парова, а то је $O(n^2)$.

```
#include <iostream>
#include <vector>

using namespace std;

int broj_inverzija(const vector<int>& a) {
  int n = a.size();
  int broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      if (a[j] < a[i])
        broj++;
  return broj;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  cout << broj_inverzija(a) << endl;;
  return 0;
}
```

Подели па владај - модификација алгоритма MergeSort

Размотримо како бисмо проблем решили декомпозицијом. Празан и једночлан низ немају инверзија. Ако је низ подељен на две половине, укупан број инверзија једнак је збиру броја инверзија међу елементима прве половине, броја инверзија међу елементима друге половине и броја парова елемената где први елемент припада првој, други елемент припада другој половини и први је већи од другог. Прва два броја можемо одредити рекурзивно и остаје само питање како ефикасно одредити трећи број. Да бисмо добили укупну сложеност $O(n \log n)$ тај проблем је потребно решити у сложености $O(n)$ тако да испитивање свих парова елемената из прве и друге половине не долази у обзир. Задатак би се могао лакше решити ако би прва и друга половина биле сортиране (кључни увид је да сортирање елемената тих половина не мења трећи број). Тада можемо применити технику два показивача и веома слично као у случају обједињавања два сортирана низа одредити жељени трећи број. Уместо да сортирамо половине засебно, можемо алгоритам сортирања интегрисати са бројањем инверзија и проширити инваријанту наше функције тако да током бројања инверзија уједно сортира низ. На основу инваријанте, рекурзивни позиви ће сортирати леву и десну половину, а да бисмо је одржали, током одређивања трећег броја вршићемо обједињавање сортираних низова (исто као у алгоритму MergeSort).

```cpp
#include <iostream>
#include <vector>

using namespace std;

int broj_inverzija(vector<int>& a, int l, int d, vector<int>& b) {
  if (l >= d)
    return 0;
  int s = l + (d - l) / 2;
  int broj = 0;
  broj += broj_inverzija(a, l, s, b);
  broj += broj_inverzija(a, s+1, d, b);
  int pl = l, pd = s+1, pb = 0;
  while (pl <= s && pd <= d) {
    if (a[pl] <= a[pd])
      b[pb++] = a[pl++];
    else {
      broj += s - pl + 1;
      b[pb++] = a[pd++];
    }
  }
  while (pl <= s)
    b[pb++] = a[pl++];
  while (pd <= d)
    b[pb++] = a[pd++];

  copy(begin(b), next(begin(b), d - l + 1), next(begin(a), l));

  return broj;
}

int broj_inverzija(const vector<int>& a) {
  vector<int> tmp1(a.size()), tmp2(a.size());
  copy(begin(a), end(a), begin(tmp1));
  return broj_inverzija(tmp1, 0, a.size()-1, tmp2);
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  cout << broj_inverzija(a) << endl;;
  return 0;
}
```

Фенвиково дрво

Један начин да се задатак ефикасно реши је да одржавамо структуру података која ефикасно може уметнути нови елемент и дати одговор на то колико је елемената у структури строго мање од датог. Једна таква структура је Фенвиково дрво. Низ обилазимо са десног краја, за сваки елемент проверавамо колико је елемената у дрвету веће од њега, увећавамо бројач инверзија за тај број и након тога умећемо елемент у дрво (обилазак са десног краја је потребан јер нам Фенвиково дрво једноставно даје одговор на то колико је елемената мање од датог, али не и колико је елемената веће од датог). Пошто величина дрвета зависи од вредности највећег елемента у њему, а нама за број инверзија нису важне апсолутне вредности елемената, него само њихов међусобни однос, пре примене алгоритма можемо сваки елемент заменити са његовим рангом у сортираном низу (исти елементи могу да имају исти ранг). Ово је могуће урадити на било који начин који је описан у задатку

Ранг сваког елемента.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void dodaj(vector<int>& drvo, int k, int v) {
  while (k < drvo.size()) {
    drvo[k] += v;
    k += k & -k;
  }
}

int zbirPrefiksa(const vector<int>& drvo, int k) {
  int zbir = 0;
  while (k > 0) {
    zbir += drvo[k];
    k -= k & -k;
  }
  return zbir;
}

int brojInverzija(const vector<int>& a) {
  int n = a.size();
  vector<int> b = a;
  sort(begin(b), end(b));
  vector<int> drvo(n+1, 0);
  int broj = 0;
  for (int i = n-1; i >= 0; i--) {
    int x = distance(begin(b), lower_bound(begin(b), end(b), a[i])) + 1;
    broj += zbirPrefiksa(drvo, x-1);
    dodaj(drvo, x, 1);
  }
  return broj;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  cout << brojInverzija(a) << endl;;
  return 0;
}
```

Задатак: Број елемената десно од датог елемента већих или једнаких од њега

Дат је низ од n целих бројева који представља вредност акција једне компаније током одређеног временског периода. Дан d (дане бројимо од 0 до $n-1$) сматрамо k-успешним ако се након њега није јавило строго више од k дана у којима је вредност акција била већа или једнака вредности током дана d. Напиши програм који за разне вредности d и k одређује да ли је дан d био k-успешан.

Улаз: Са стандардног улаза се учитава број дана n ($1 \leq n \leq 50000$), а затим у наредном реду n различитих

целих бројева између 1 и 10^6 који представљају вредности акција компаније. Након тога се учитава број питања q ($1 \leq q \leq 50000$), а затим у q наредних редова по два броја d ($0 \leq d < n$) и k ($0 \leq k \leq n$), раздвојена размаком.

Излаз: За сваки од q упита на стандардни излаз исписати `da` или `ne` у зависности од тога да ли је дан d k-успешан.

Пример

Улаз	*Излаз*
7 30 14 20 15 8 23 25 1 1 2	ne

Решење

Решење грубом силом подразумева да сваки пут изнова пребројимо елементе иза позиције d који су већи од броја a_d (линеарним проласком кроз тај део низа). Сложеност овог решења је $O(q \cdot n)$, где је q број упита, а n дужина низа. Решење се може свести на $O(n^2)$, ако се пребројавање унапред изврши за сваку позицију и резултати сместе у помоћни низ.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ubrzavamo ulaz i izlaz
  ios_base::sync_with_stdio(false); cin.tie(0);
  // ucitavamo elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  // obradjujemo q upita
  int q;
  cin >> q;
  for (int i = 0; i < q; i++) {
    // izracunavamo broj elemenata desno od a[d] vecih ili jednakih od njega
    int d, k;
    cin >> d >> k;
    int brojVecih = 0;
    for (int j = d + 1; j < n; j++)
      if (a[j] >= a[d])
        brojVecih++;
    // proveravamo da li je takvih elemenata bar k
    if (brojVecih > k)
      cout << "ne\n";
    else
      cout << "da\n";
  }
  return 0;
}
```

Решење може бити засновано на принципу подели-па-владај, модификацијом алгоритма сортирања обједињавањем (слично као у задатку Број инверзија).

Ако је низ једночлан, тада је он сортиран и не постоје ни мањи ни већи елементи од тог датог.

У супротном, претпоставимо да је низ подељен на две (непразне) половине приближно једнаке дужине. Претпоставимо да је након рекурзивног позива свака половина сортирана и да за сваки елемент знамо колико у

тој половини низа има елемената десно од њега који су већи или једнаки од њега. На пример, ако је полазни низ `4 1 5 8 5 3 6`, добијамо

```
niz:            1 4 5 8   3 5 6
vecih_desno:    2 2 1 0   1 1 0
```

Након тога прелазимо на обједињавање два сортирана низа и ажурирање података за цео низ. Уобичајено, користимо два показивача (који указују на текући елемент у свакој половини) и помоћни низ у који смештамо резултат. Користимо и помоћни низ у који смештамо тражене резултујуће податке. Поредимо два текућа елемента.

- Ако је текући елемент леве половине мањи или једнак од текућег елемента десне половине, преписујемо га у резултујући сортирани низ. Сви елементи у десној половини су десно од њега у полазном низу. Од њих, они који су мањи од текућег елемента леве половине су већ преписани у резултујући низ, док су сви преостали елементи десне половине већи или једнаки од њега. Зато се укупан број елемената који су десно од текућег и већи или једнаки су од њега може добити сабирањем таквих елемената из леве половине низа (који знамо на основу рекурзивног позива) и броја преосталих елемената у десној половини низа (који лако израчунавамо одузимањем позиције текућег елемента у десној половини, од њеног укупног броја елемената).
- Ако је текући елемент десне половине већи од текућег елемента леве половине, онда њега преписујемо у резултујући сортирани низ. Број елемената који су у оригиналном низу десно од њега и већи или једнаки су од њега, један је броју таквих елемената у десној половини (јер у левој половини нема елемената који су десно од текућег), па само преписујемо податак који знамо на основу рекурзивног позива.

У текућем примеру, добијамо следећи резултат:

```
niz:            1   3  4    5    5  6  8
vecih_desno:    2+3 1  2+2  1+2  1  0  0
```

тј.

```
niz:            1  3  4  5  5  6  8
vecih_desno:    5  1  4  3  1  0  0
```

Сложеност једног корака обједињавања две половине је линеарна у односу на укупну дужину те две половине, па је сложеност целог поступка сортирања и израчунавања иста као у случају сортирања обједињавањем и износи $O(n \log n)$.

Један проблем који имамо је то што тражене податке за сваки елемент чувамо у низу, али уређене у односу на сортирани редослед бројева из оригиналног низа. Да бисмо те податке добили у уређене у односу на оригинални редослед елемената полазног низа, можемо чувати пермутацију елемената приликом сортирања (за сваки позицију у резултујућем сортирамо низу чувамо позицију елемента у оригиналном низу који након сортирања завршава на тој позицији).

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

// rekurzivna funkcija koja:
// - popunjava niz veciDesno koji za svaki element x u delu niza [l, d]
//   izracunava broj elemenata u delu [l, d] koji su desno od x i veci od x
// - kao propratni efekat sortira niz
// - u niz perm upisuje permutaciju niza koja sluzi da se nakon zavrsetka
//   brojevi vecih elemenata mogu pravilno rasporediti na odgovarajuce pozicije
// Tmp su pomocni nizovi
void prebrojVeceDesno(vector<int>& a, int l, int d, vector<int>& aTmp,
                      vector<int>& veciDesno, vector<int>& veciDesnoTmp,
                      vector<int>& perm, vector<int>& permTmp) {
  // ako niz ima bar dva elementa
```

```
  if (d - l >= 1) {
    // delimo niz u delove [l, s] i [s+1, d]
    int s = l + (d - l) / 2;

    // rekurzivno obradjujemo obe polovine
    prebrojVeceDesno(a, l, s, aTmp, veciDesno, veciDesnoTmp, perm, permTmp);
    prebrojVeceDesno(a, s+1, d, aTmp, veciDesno, veciDesnoTmp, perm, permTmp);

    // vrsimo objedinjavanje dva sortirana niza, popunjavajuci pri tom
    // za svaki element podatke o tome koliko u originalnom nizu ima
    // elemenata koji su veći ili jednaki od njega, a nalaze se desno od njega
    // prvo se sve smesta u pomocne nizove
    int i = l, j = s+1, k = 0;
    while (i <= s || j <= d)
      if (i <= s && (j > d || a[i] <= a[j])) {
        veciDesnoTmp[k] = veciDesno[i] + (d - j + 1);
        permTmp[k] = perm[i];
        aTmp[k++] = a[i++];
      } else {
        veciDesnoTmp[k] = veciDesno[j];
        permTmp[k] = perm[j];
        aTmp[k++] = a[j++];
      }

    // kopiramo vrednosti iz pomocnih nizova u glavne
    copy(begin(aTmp), next(begin(aTmp), d-l+1),
         next(begin(a), l));
    copy(begin(veciDesnoTmp), next(begin(veciDesnoTmp), d-l+1),
         next(begin(veciDesno), l));
    copy(begin(permTmp), next(begin(permTmp), d-l+1),
         next(begin(perm), l));
  }
}

// za svaki element u nizu se odredjuje broj elemenata koji su desno
// od njega i veci su od njega
vector<int> prebrojVeceDesno(vector<int>& a) {
  int n = a.size();
  vector<int> tmp(n);
  vector<int> veciDesno(n, 0);
  vector<int> veciDesnoTmp(n);
  // permutacija koja se koristi tokom sortiranja
  vector<int> perm(n);
  iota(begin(perm), end(perm), 0);
  vector<int> permTmp(n);
  prebrojVeceDesno(a, 0, n-1, tmp, veciDesno, veciDesnoTmp, perm, permTmp);
  // rasporedjujemo izracunate brojace na odgovarajuce pozicije
  for (int i = 0; i < n; i++)
    veciDesnoTmp[perm[i]] = veciDesno[i];
  // vracamo konacan rezultat
  return veciDesnoTmp;
}

int main() {
  // ucitavamo elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
```

```
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // za svaki element odredjujemo broj elemenata desno od njega koji
  // su veci od njega
  vector<int> vecihDesno = prebrojVeceDesno(a);

  // obradjujemo q upita
  int q;
  cin >> q;
  for (int i = 0; i < q; i++) {
    int d, k;
    cin >> d >> k;
    if (vecihDesno[d] > k)
      cout << "ne\n";
    else
      cout << "da\n";
  }

  return 0;
}
```

Задатак можемо решити и употребом Фенвикових или сегментних дрвета. Пошто нам је битан само релативан однос елемената, а да бисмо смањили заузеће меморије у низу бројача који ћемо користити, низ препроцесирамо тако што за сваки елемент одређујемо позицију тог елемента у сортираном низу (сви елементи који се понављају имају исту позицију).

На пример, уместо низа

```
4 1 5 8 5 3 6
```

разматрамо низ

```
2 0 3 6 3 1 5
```

Идеја је да елементе овако добијеног низа индекса обилазимо здесна на лево и да за сваку позицију одржавамо бројач елемената на тој позицији које смо до тог тренутка обишли. Иницијално су сви бројачи једнаки 0. Пре обраде елемената, број елемената који су већи или једнаки од њега и десно су од њега добијамо сабирањем вредности бројача од тог елемента па надесно (израчунавањем збира суфикса низа бројача).

```
0 1 2 3 4 5 6
0 0 0 0 0 0 0
```

Обрађујемо индекс 5. Збир елемената на позицији 5 и 6 је 0, што значи да иза последњег елемента у низу (то је елемент 6) нема елемената већих или једнаких од њега. Увећавамо бројач на позицији 5.

```
0 1 2 3 4 5 6
0 0 0 0 0 1 0
```

Обрађујемо затим индекс 1. Збир елемената на позицијама од 1 до 6 је 1, што значи да десно од претпоследњег елемента (то је елемент 3) постоји један елемент већи или једнак од њега. Увећавамо бројач на позицији 1.

```
0 1 2 3 4 5 6
0 1 0 0 0 1 0
```

Обрађујемо затим индекс 3. Збир елемената на позицијама од 3 до 6 је 1, што значи да десно од елемента 5 у полазном низу постоји један елемент већи или једнак од њега. Увећавамо бројач на позицији 3.

```
0 1 2 3 4 5 6
0 1 0 1 0 1 0
```

Обрађујемо затим индекс 6. Збир елемената на позицијама од 6 до 6 је 0, што значи да десно од елемента 8 у полазном низу не постоји ни један елемент већи или једнак од њега. Увећавамо бројач на позицији 6.

```
0 1 2 3 4 5 6
0 1 0 1 0 1 1
```

Обрађујемо затим индекс 3. Збир елемената на позицијама од 3 до 6 је 3, што значи да десно од елемента 5 у полазном низу постоји 3 елемента који су већи или једнаки од њега. Увећавамо бројач на позицији 3.

```
0 1 2 3 4 5 6
0 1 0 2 0 1 1
```

Обрађујемо затим индекс 0. Збир елемената на позицијама од 0 до 6 је 5, што значи да десно од елемента 1 у полазном низу постоји 5 елемента који су већи или једнаки од њега. Увећавамо бројач на позицији 0.

```
0 1 2 3 4 5 6
1 1 0 2 0 1 1
```

Обрађујемо затим индекс 2. Збир елемената на позицијама од 2 до 6 је 4, што значи да десно од елемента 4 у полазном низу постоји 4 елемента који су већи или једнаки од њега. Увећавамо бројач на позицији 2.

```
0 1 2 3 4 5 6
1 1 1 2 0 1 1
```

Да бисмо брзо могли да израчунавамо вредност збира суфикса низа бројача елементе ћемо чувати у Фенвиковом дрвету и то тако да уместо бројача на позицији i користимо бројач на позицији $n - i$, чиме се проблем одређивања збира суфикса своди на проблем одређивања збира префикса.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// uvecavanje elementa u Fenvikovom drvetu na poziciji k za vrednost v
void dodaj(vector<int>& drvo, size_t k, int v) {
  while (k < drvo.size()) {
    drvo[k] += v;
    k += k & -k;
  }
}

// odredjivanje zbira prefiksa Fenvikovog drveta zakljucno sa pozicijom k
int zbirPrefiksa(const vector<int>& drvo, int k) {
  int zbir = 0;
  while (k > 0) {
    zbir += drvo[k];
    k -= k & -k;
  }
  return zbir;
}

// za svaki element u nizu se odredjuje broj elemenata koji su desno
// od njega i veci su od njega
vector<int> prebrojVeceDesno(vector<int>& a) {
  int n = a.size();
  // racunamo rang svakog elementa
  vector<int> b = a;
  sort(begin(b), end(b));
  for (int i = 0; i < n; i++)
    a[i] = distance(begin(b), lower_bound(begin(b), end(b), a[i]));

  vector<int> drvo(n+1, 0);
  vector<int> veciDesno(n, 0);
  for (int i = n-1; i >= 0; i--) {
    veciDesno[i] = zbirPrefiksa(drvo, n - a[i]);
    dodaj(drvo, n - a[i], 1);
  }
```

```
  return veciDesno;
}


// 4 1 5 8 5 3 6
// 4 0 2 6 2 1 5
// 3 7 5 1 5 6 2

// 1 2 3 4 5 6 7
// 0 0 0 0 0 1 0

int main() {
  // ucitavamo elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // za svaki element odredjujemo broj elemenata desno od njega koji
  // su veci od njega
  vector<int> vecihDesno = prebrojVeceDesno(a);

  // obradjujemo q upita
  int q;
  cin >> q;
  for (int i = 0; i < q; i++) {
    int d, k;
    cin >> d >> k;
    if (vecihDesno[d] > k)
      cout << "ne\n";
    else
      cout << "da\n";
  }

  return 0;
}
```

Задатак: Број сортираних тројки

Написати програм којим се у датом низу a целих бројева одређује колико постоји тројки $i < j < k$ таквих да је $a_i < a_j < a_k$.

Улаз: У првој линији стандардног улаза налази се број елемената низа n ($1 \leq n \leq 50000$), а у следећих n линија налази се редом елементи низа a (цели бројеви између 0 и 50000).

Излаз: На стандардном излазу у једној линији приказати тражени број тројки - ако је тај број већи од 10^9 приказати му последњих 9 цифара (без почетних нула).

Пример 1

Улаз	*Излаз*
5 4 1 5 3 8	3

Објашњење

Trojke su 4, 5, 8, zatim 1, 5, 8 i na kraju 1, 3, 8.

Пример 2

Улаз

```
5
1
2
3
4
5
```

Излаз

```
10
```

Пример 3

Улаз

```
5
5
4
3
2
1
```

Излаз

```
0
```

Решење

Груба сила

Задатак можемо решити анализирањем свих тројки a_i, a_j, a_k таквих да је $i < j < k$. То постижемо коришћењем 3 бројачка циклуса, и то редом, први циклус којим бројач i узима вредности од 0 до $n-3$, други циклус којим бројач j узима вредности од $i+1$ до $n-2$ и трећи циклус којим бројач k узима вредности од $j+1$ до $n-1$ (угнежђене петље овог типа смо видели, на пример, у задацима и).

Пребројавамо све тројке које задовољавају дати услов. Ако је $a_i < a_j < a_k$ бројач уређених тројки увећамо за 1. Пошто број тројки може бити превелики тражи се остатак при дељењу тог броја са 10^9. Овај остатак је могуће израчунати на крају тела спољашње петље. Наиме, израчунавање остатка у сваком кораку би непотребно значајно успорило програм, а број елемената низа је такав да увећање бројача током једног извршавања две унутрашње петље не може проузроковати прекорачење опсега целих бројева.

Приметимо да је број поређења веома велики - он кубно зависи од броја елемената низа.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ucitavamo elemente niza
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
      cin >> a[i];

  // proveravamo i brojimo sve trojke
  long long int sortiranihTrojki = 0;
  for (int i = 0; i < n - 2; i++) {
    for (int j = i + 1; j < n - 1; j++)
```

```
      for (int k = j + 1; k < n; k++)
        if (a[i] < a[j] && a[j] < a[k])
          sortiranihTrojki++;
    const int mod = 1e9;
    sortiranihTrojki %= mod;
  }

  // ispisujemo broj trojki
  cout << sortiranihTrojki << endl;

  return 0;
}
```

Једна могућа оптимизација настаје када се примети да нам трећи циклус није потребан у случају када је $a_i \geq a_j$. Дакле, у телу другог циклуса се проверава да ли је $a_i < a_j$ и трећи циклус се извршава само ако је овај услов испуњен. Ипак, овим се и даље задржава кубна сложеност у најгорем случају (то је случај сортираног низа).

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ucitavamo elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
      cin >> a[i];

  // proveravamo i brojimo sve trojke
  long long int sortiranihTrojki = 0;
  for (int i = 0; i < n - 2; i++) {
    for (int j = i + 1; j < n - 1; j++)
      // ako a[i] nije manje od a[j], treci broj ne treba proveravati
      if (a[i] < a[j])
        for (int k = j + 1; k < n; k++)
          if (a[j] < a[k])
            sortiranihTrojki++;
    int mod = 1e9;
    sortiranihTrojki %= mod;
  }

  // ispisujemo broj trojki
  cout << sortiranihTrojki << endl;
  return 0;
}
```

Анализа сваког средишњег елемента тројке

Анализирајмо нешто ефикаснији приступ решењу задатка. Основна идеја је да за сваки елемент a_j где је $0 < j < n - 1$, одредимо број уређених тројки код којих је a_j средњи елемент (први и последњи елемент низа не могу бити средњи елементи неке тројке). Елемент a_j је средњи елемент уређене тројке $a_i < a_j < a_k$, за сваки елемент a_i лево од њега који је мањи од њега, и за сваки елемент a_k десно од њега који је већи од њега. Да би одредили број уређених тројки код којих је a_j средњи елемент потребно је одредити:

- $brojManjihLevo$ - број елемената лево од елемента a_j (елемената a_i за $0 \leq \ < i$) који су мањи од њега ($a_i < a_j$),
- $brojVecihDesno$ - број елемената десно од елемента a_j (a_k за $j < k < n$) који су већи од њега ($a_k >$

a_j).

Тада број уређених тројки у којима је a_j средњи елемент добијамо као прозвод $brojManjihLevo \cdot brojVecihDesno$. Тада укупан број тројки увећавамо за овај број.

При том, потребно је примењивати аритметику по модулу. О њеним општим правилима било је речи у задатку Операције по модулу, али ипак можемо извршити неке ситне оптимизације (које, заправо и нису критичне за брзину коначног програма). Пошто можемо претпоставити да је претходна вредност укупног броја већ мања од 10^9 пре његовог увећавања нема потребе израчунавати његов остатак по модулу m тј. уместо правла $br = (br \bmod m + p \bmod m) \bmod m$ можемо само применити $br = (br + p \bmod m) \bmod m$. Додатно, пошто пошто на основу ограничења у задатку важи да ће производ p бити мањи од $(2,5 \cdot 10^4)^2 = 6,25 \cdot 10^8$, што је мање од 10^9 ни производ p бројева $brojManjihLevo$ и $brojVecihDesno$ није неопходно рачунати по модулу (чак ни ако се користи 32-битни тип података). Наиме, производ је максималан ако су сабирци једнаки.

Приметимо да је временска сложеност овог алгоритма квадратна, а не кубна, што је веома значајна уштеда, али, могуће је конструисати и ефикаснији алгоритам од тога.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ucitavamo elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ukupan broj sortiranih trojki
  int sortiranihTrojki = 0;
  // obradjujemo jedan po jedan element niza
  for (int j = 1; j < n - 1; j++) {
    // odredjujemo broj elemenata levo od tekuceg manjih od njega
    int manjihLevo = 0;
    for (int i = 0; i < j; i++)
      if (a[i] < a[j])
        manjihLevo++;

    // odredjujemo broj elemenata desno od tekuceg vecih od njega
    int vecihDesno = 0;
    for (int k = j + 1; k < n; k++)
      if (a[j] < a[k])
        vecihDesno++;

    // uvecavamo brojac za ukupan broj trojki u kojem je tekuci element srednji
    const int mod = 1e9;
    sortiranihTrojki = (sortiranihTrojki + manjihLevo * vecihDesno) % mod;
  }

  // ispisujemo ukupan broj sortiranih trojki
  cout << sortiranihTrojki << endl;
  return 0;
}
```

Пребројавање мањих и већих елемената испред и иза датог се може извршити и библиотечким функцијама.

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>
using namespace std;

int main() {
  // ucitavamo elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ukupan broj sortiranih trojki
  int sortiranihTrojki = 0;
  // obradjujemo jedan po jedan element niza
  for (int j = 1; j < n - 1; j++) {
    // odredjujemo broj elemenata levo od tekuceg manjih od njega
    int manjihLevo = count_if(a.begin(), next(a.begin(), j),
                              [&](int x) {
                                return x < a[j];
                              });
    // odredjujemo broj elemenata desno od tekuceg vecih od njega
    int vecihDesno = count_if(next(a.begin(), j+1), a.end(),
                              [&](int x) {
                                return x > a[j];
                              });
    // uvecavamo brojac za ukupan broj trojki u kojem je tekuci element srednji
    const int mod = 1e9;
    sortiranihTrojki = (sortiranihTrojki + manjihLevo * vecihDesno) % mod;
  }

  // ispisujemo ukupan broj sortiranih trojki
  cout << sortiranihTrojki << endl;

  return 0;
}
```

Сортирање обједињавањем

Кључни део решења је за сваки елемент низа одредити број елемената који се у низу налазе лево од њега и строго су мањи од њега и број елемената који се у низу налазе десно од њега и строго су већи од њега. То је могуће ефикасно урадити техником "подели-па-владај", модификацијом алгоритма сортирања обједињавањем (слично као у задатку Број инверзија). Једноставности ради, претпоставимо прво да су сви елементи у низу различити.

Ако је низ једночлан, тада је он сортиран и не постоје ни мањи ни већи елементи од тог датог.

У супротном, претпоставимо да је низ подељен на две (непразне) половине приближно једнаке дужине. Претпоставимо да је након рекурзивног позива свака половина сортирана и да за сваки елемент знамо колико у тој половини низа има мањих елемената лево и већих елемената десно од њега. На пример, ако је полазни низ `4 1 5 3 8`, добијамо

```
niz:            1 4 5   3 8
manjih_levo:    0 0 2   0 1
vecih_desno:    1 1 0   1 0
```

Након тога прелазимо на обједињавање два сортирана низа и ажурирање тражених података за цео низ. Уобичајено, користимо два показивача (који указују на текући елемент у свакој половини) и помоћни низ у који смештамо резултат. Користимо и два помоћна низа у које смештамо резултујуће податке о броју мањих и већих елемената. Поредимо два текућа елемента.

- Ако је текући елемент леве половине мањи од текућег елемента десне половине, преписујемо га у ре-

зултујући сортирани низ. Елементи лево од њега су само они у левој половини, па се број елемената који су лево од њега и мањи од њега само преписује из леве половине. Сви елементи у десној половини су десно од њега у полазном низу. Од њих, они који су мањи од текућег елемента леве половине су већ преписани у резултујући низ, док су сви преостали елементи десне половине већи од њега. Зато се укупан број елемената који су десно и од текућег и већи су од њега може добити сабирањем таквих елемената из леве половине низа (који знамо на основу рекурзивног позива) и броја преосталих елемената у десној половини низа (који лако израчунавамо одузимањем позиције текућег елемента у десној половини, од њеног укупног броја елемената).

- Ако је текући елемент десне половине већи од текућег елемента леве половине, онда њега преписујемо у резултујући сортирани низ. Укупан број елемената који су лево од њега и мањи су од њега се добија сабирањем броја елемената у десној половини који су лево од текућег и мањи су од њега (а тај број знамо на основу резултата рекурзивног позива) и броја преписаних елемената леве половине (јер су сви елементи у левој половини лево од текућег елемента десне половине, док су од њега мањи тачно они који су већ преписани). Што се тиче броја елемената који су десно од текућег елемента десне половине и већи су од њега, ту само преписујемо податак о елементима из десне половине (јер у левој половини нема елемената који су десно од текућег).

У текућем примеру, добијамо следећи резултат:

```
niz:            1   3   4    5    8
manjih_levo:    0   0+1 0    2    1+3
vecih_desno:    1+2 1   1+1  0+1  0
```

тј.

```
niz:            1  3  4  5  8
manjih_levo:    0  1  0  2  4
vecih_desno:    3  1  2  1  0
```

Ако елементи нису различити, алгоритам је потребно још мало дорадити.

- Када се преписује текући елемент десне половине, потребно је обратити пажњу да су у левој половини обрађени сви елементи који су мањи или једнаки од њега, па се за број елемената у левој половини који су строго мањи од њега не може узети број преписаних елемената, већ је из тог броја потребно искључити оне преписане елементе који су једнаки текућем. То се лако решава увођењем једног "заосталог" показивача који ће увек да указује тачно иза елемената леве половине који су строго мањи од текућег елемента десне половине).

- Аналогно, када се преписује елемент леве половине, потребно је да знамо и елементе десне половине који су строго већи од њега, а то нису сви они иза текућег (јер неки од њих могу бити и једнаки текућем). То се лако решава увођењем једног "побеглог" показивача који ће увек да показује тачно иза свих елемената десне половине који су већи или једнаки од текућег елемента леве половине.

Сложеност једног корака обједињавања две половине је линеарна у односу на укупну дужину те две половине, па је сложеност целог поступка сортирања и израчунавања иста као у случају сортирања обједињавањем и износи $O(n \log n)$.

Број тројки се лако добија израчунавањем скаларног производа добијених низова (за шта је довољно $O(n)$ операција).

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// rekurzivna funkcija koja:
// - popunjava niz manjihLevo koji za svaki element x u delu niza [l, d]
//   izracunava broj elemenata u delu [l, d] koji su levo od x i manji od x
// - popunjava niz vecihDesno koji za svaki element x u delu niza [l, d]
//   izracunava broj elemenata u delu [l, d] koji su desno od x i veci od x
// - kao propratni efekat sortira niz
// Tmp su pomocni nizovi
void izbroj(vector<int>& a, int l, int d, vector<int>& aTmp,
```

```
            vector<int>& manjihLevo, vector<int>& manjihLevoTmp,
            vector<int>& vecihDesno, vector<int>& vecihDesnoTmp) {
  if (d - l >= 1) {
    // delimo niz u delove [l, s] i [s+1, d]
    int s = l + (d - l) / 2;

    // rekurzivno obradjujemo obe polovine
    izbroj(a, l, s, aTmp,
           manjihLevo, manjihLevoTmp,
           vecihDesno, vecihDesnoTmp);
    izbroj(a, s+1, d, aTmp,
           manjihLevo, manjihLevoTmp,
           vecihDesno, vecihDesnoTmp);

    // vrsimo objedinjavanje dva sortirana niza, popunjavajuci
    // pri tom podatke o elementima manjim levo i vecim desno
    // prvo se sve smesta u pomocne nizove
    int i = l, i0 = l, j = s+1, j0 = s + 1, k = 0;
    while (i <= s || j <= d)
      if (i <= s && (j > d || a[i] <= a[j])) {
        manjihLevoTmp[k] = manjihLevo[i];
        while (j0 <= d && a[j0] <= a[i])
          j0++;
        vecihDesnoTmp[k] = vecihDesno[i] + (d - j0 + 1);
        aTmp[k++] = a[i++];
      } else {
        while(i0 < i && a[i0] < a[j])
          i0++;
        manjihLevoTmp[k] = (i0 - l) + manjihLevo[j];
        vecihDesnoTmp[k] = vecihDesno[j];
        aTmp[k++] = a[j++];
      }

    // kopiramo vrednosti iz pomocnih nizova u glavne
    copy(begin(aTmp), next(begin(aTmp), d-l+1),
         next(begin(a), l));
    copy(begin(manjihLevoTmp), next(begin(manjihLevoTmp), d-l+1),
         next(begin(manjihLevo), l));
    copy(begin(vecihDesnoTmp), next(begin(vecihDesnoTmp), d-l+1),
         next(begin(vecihDesno), l));
  }
}

// ulazna tacka u rekurzivnu funkciju
void izbroj(vector<int>& a, vector<int>& manjihLevo, vector<int>& vecihDesno) {
  vector<int> aTmp(a.size());
  vector<int> manjihLevoTmp(a.size());
  vector<int> vecihDesnoTmp(a.size());
  izbroj(a, 0, a.size() - 1, aTmp,
         manjihLevo, manjihLevoTmp,
         vecihDesno, vecihDesnoTmp);
}

int main() {
  // ucitavamo elemente u niz
  int n;
  cin >> n;
  vector<int> a(n);
```

```
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // za svaki element niza izracunavamo broj elemenata levo od njega
  // koji su manji od njega i broj elemenata desno od njega koji su
  // veci od njega
  vector<int> manjihLevo(n, 0), vecihDesno(n, 0);
  izbroj(a, manjihLevo, vecihDesno);

  // izracunavamo i ispisujemo broj sortiranih trojki
  const int mod = 1e9;
  int sortiranihTrojki = 0;
  // analiziramo svaki elementi koji moze biti sredisnji element trojke
  for (int i = 1; i < n-1; i++)
    sortiranihTrojki = (sortiranihTrojki + manjihLevo[i]*vecihDesno[i]) % mod;
  cout << sortiranihTrojki << endl;

  return 0;
}
```

Фенвиково дрво

Број елемената лево од датог елемента и мањих од њега, као и број елемената десно од датог елемента и већих од њега се може одредити помоћу структуре података која допушта ефикасно постављање вредности елемента на датој позицији низа фиксне величине и израчунавање вредности збира префикса до неке позиције у том низу. То може бити, на пример, сегментно или Фенвиково стабло.

Ако одређујемо број елемената лево од датог који су строго мањи од њега, обилазићемо низ слева надесно и одржаваћемо скуп раније обрађених елемената (ако се елементи понављају, онда је то заправо мултискуп). Он је иницијално празан и проширује се текућим елементом, чим га обрадимо. Тај мултискуп ће бити представљен низом бројева такав да се на позицији i налази број појављивања елемента i у том мултискупу садржи вредност i, при чему ће над тим низом бити изграђено Фенвиково (или сегментно) дрво. Тада се број елемената строго мањих од датог елемента i може одредити одређивањем збира префикса испред позиције i (што је операција логаритамске сложености ако се користи неко од поменутих дрвета). Пошто величина дрвета зависи од вредности највећег елемента у њему, а нама нису важне апсолутне вредности елемената, него само њихов међусобни однос, пре примене алгоритма можемо сваки елемент заменити са његовим рангом у сортираном низу (исти елементи могу да имају исти ранг). Ово је могуће урадити на било који начин који је описан у задатку Ранг сваког елемента.

Ако анализирамо елементе веће од датог, тада низ треба обилазити здесна налево. Ако нас интересују елементи већи, а не мањи од датог, онда је потребно применити неку функцију која обрће број елемената (на пример, уместо елемента i може се посмтрати $n-i+1$, чиме интервал елемената од $[1,n]$ остаје непромењен).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// uvecavanje elementa u Fenvikovom drvetu na poziciji k za vrednost v
void dodaj(vector<int>& drvo, size_t k, int v) {
  while (k < drvo.size()) {
    drvo[k] += v;
    k += k & -k;
  }
}

// odredjivanje zbira prefiksa Fenvikovog drveta zakljucno sa pozicijom k
int zbirPrefiksa(const vector<int>& drvo, size_t k) {
  int zbir = 0;
```

```
  while (k > 0) {
    zbir += drvo[k];
    k -= k & -k;
  }
  return zbir;
}

int main() {
  // ucitavamo elemente
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // odredjujemo rang svakog elementa
  vector<int> b = a;
  sort(begin(b), end(b));
  for (int i = 0; i < n; i++)
    a[i] = distance(begin(b), lower_bound(begin(b), end(b), a[i])) + 1;

  // pomocu Fenvikovog drveta za svaki element niza odredjujemo
  // broj elemenata koji su levo od njega i strogo su manji od njega
  vector<int> drvo(n);
  vector<int> manjihLevo(n);
  for (int i = 0; i < n; i++) {
    manjihLevo[i] = zbirPrefiksa(drvo, a[i]-1);
    dodaj(drvo, a[i], 1);
  }

  // pomocu Fenvikovog drveta za svaki element niza odredjujemo
  // broj elemenata koji su desno od njega i strogo su veci od njega
  fill(begin(drvo), end(drvo), 0);
  vector<int> vecihDesno(n);
  for (int i = n-1; i >= 0; i--) {
    vecihDesno[i] = zbirPrefiksa(drvo, n - a[i]);
    dodaj(drvo, n - a[i] + 1, 1);
  }

  // odredjujemo broj sortiranih trojki elemenata niza
  const int mod = 1e9;
  int sortiranihTrojki = 0;
  for (int i = 0; i < n; i++)
    sortiranihTrojki = (sortiranihTrojki + manjihLevo[i]*vecihDesno[i]) % mod;
  cout << sortiranihTrojki << endl;
}
```

Задатак: Бинарне слике

Бинарна слика је слика која садржи само две боје, на пример црну и белу.

Садржај бинарне слике квадратног облика (код које је број редова једнак броју колона) може да се зада помоћу стринга на следећи начин:

- ако је цела слика бела, она се описује стрингом “1”
- ако је цела слика црна, она се описује стрингом “0”
- ако слика има и црне и беле пикселе, стринг који је описује почиње отвореном угластом заградом [, за којом следе описи горње десне, горње леве, доње леве и доње десне четвртине (слепљено, тј. без икаквих симбола између описа делова), а на крају стринга је затворена угласта заграда].

На пример, стринг “[1110]” описује слику величине 2×2, која у доњем десном углу има црни пиксел, док су остала три пиксела бела. Слично томе, стринг “[00[0010]0]” описује слику величине 4×4, која у доњем левом углу има бели пиксел, док су осталих 15 пиксела црни.

Над пикселима дефинишемо операције уније и пресека н аследећи начин:

Унија два пиксела је црни пиксел ако и само ако су оба та пиксела црна, а у противном је унија бели пиксел.

Пресек два пиксела је бели пиксел ако и само ако су оба та пиксела бела, а у противном је пресек црни пиксел.

Унија две квадратне слике је слика чији су пиксели уније одговарајућих пиксела у сликама над којима се рачуна унија. Аналогно се дефинише и пресек две слике.

Напиши прогам који учитава описе две бинарне квадратне слике, а исписује описе њихове уније и пресека.

Улаз: Са стандардног улаза се учитавају две ниске, свака у посебном реду. Ниске представљају описе две бинарне слике квадратног облика. Дужине ниски не прелазе $10\,000$.

Излаз: На стандардни излаз у први ред исписати опис уније двеју датих слика, а у други ред опис њихвоог пресека.

Пример

Улаз	*Излаз*
`[[0010]1[0110]0]`	`[11[1110][1100]]`
`[1[0001][1000][1100]]`	`[[0010][0001]00]`

Решење

Основни случај је да се опис једне од датих слика састоји од само једног карактера. У том случају лако долазимо до резултата:

- Ако је опис једне слике “1”, опис уније је “1”, а опис пресека је исти као опис друге слике.
- Ако је опис једне слике “0”, опис уније је исти као опис друге слике, а опис пресека је “0”.

Преостали случај је да ни једна слика није једнобојна. Тада је потребно издвојити описе појединих четвртина слика, применити опрерације на парове одговарајућих четвртина и на крају спојити добијене описе и по потреби их “укрупнити” - ако су све 4 четвртине у резултујућој слици једнобојне и исте боје, опис може (и треба) да се поједностави.

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

string slA, slB;

vector<int> PoceciDelova(string sl, int levi, int desni) {
    vector<int> rez(4);
    int dubina = 0, iRez = 0;
    for (int i = levi+1; i < desni; i++)
    {
        if (dubina == 0) rez[iRez++] = i;
        if (sl[i] == '[') dubina++;
        else if (sl[i] == ']') dubina--;
    }
    return rez;
}

// domBoja je dominantna boja (za uniju 1, a za presek 0)
string Kombinacija(
    int aOd, int aDo, int bOd, int bDo, char domBoja) {
    if (aOd == aDo) {
        if (slA[aOd] == domBoja) return string(1, domBoja);
        else return slB.substr(bOd, bDo - bOd + 1);
```

```
    }
    if (bOd == bDo) {
        if (slB[bOd] == domBoja) return string(1, domBoja);
        else return slA.substr(aOd, aDo - aOd + 1);
    }

    vector<int> pocA = PoceciDelova(slA, aOd, aDo);
    vector<int> pocB = PoceciDelova(slB, bOd, bDo);
    string s1 = Kombinacija(pocA[0], pocA[1] - 1, pocB[0], pocB[1] - 1, domBoja);
    string s2 = Kombinacija(pocA[1], pocA[2] - 1, pocB[1], pocB[2] - 1, domBoja);
    string s3 = Kombinacija(pocA[2], pocA[3] - 1, pocB[2], pocB[3] - 1, domBoja);
    string s4 = Kombinacija(pocA[3], aDo - 1, pocB[3], bDo - 1, domBoja);
    if (s1 == "1" && s2 == "1" && s3 == "1" && s4 == "1") return "1";
    if (s1 == "0" && s2 == "0" && s3 == "0" && s4 == "0") return "0";
    return "[" + s1 + s2 + s3 + s4 + "]";
}

int main() {
  cin >> slA;
  cin >> slB;
  string unija = Kombinacija(0, slA.size() - 1, 0, slB.size() - 1, '1');
  string presek = Kombinacija(0, slA.size() - 1, 0, slB.size() - 1, '0');
  cout << unija << endl;
  cout << presek << endl;
  return 0;
}
```

Задатак: Максимални збир сегмента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Разлагање на потпроблеме

Један начин да решимо овај проблем је заснован на техници разлагања. Декомпозиција нам сугерише да је пожељно да низ поделимо на два подниза једнаке дужине чија решења можемо да конструишемо на основу индуктивне хипотезе (најчешће рекурзивним позивима). Базу и овај пут чини случај празног низа, који садржи само празан сегмент чији је збир нула. Фиксирајмо средишњи елемент низа. Све сегменте низа можемо да групишемо у три групе: сегменте који су у потпуности лево од средишњег елемента, сегменте који су у потпуности десно од средишњег елемента и сегменте који садрже средишњи елемент. Највеће збирове сегмената у првој и у другој групи знамо на основу индуктивне хипотезе. Највећи збир сегмента у трећој групи можемо лако одредити анализом свих сегмената: крећемо од једночланог сегмента који садржи само средишњи елемент и инкрементално се ширимо налево додајући један по један елемент и рачунајући текући максимум, а затим крећемо од максималног сегмента проширеног налево и инкрементално га проширујемо једним по једним елементом надесно, тражећи нови максимум.

Анализа сложености овај пут захтева комплекснији математички апарат, али је прилично једноставна када се тај апарат познаје. Наиме, ако са n означимо дужину низа $d - l + 1$ и ако време извршавања обележимо са $T(n)$, тада важи да је $T(0) = O(1)$ и да је $T(n) = 2T(n/2) + O(n)$. Наиме, врше се два рекурзивна позива за дупло мање низове, а највећи збир сегмената који обухватају средишњи елемент израчунавамо у времену $O(n)$ (што је прилично очигледно јер имамо две петље које се укупно извршавају n пута, а чија су тела константне сложености). На основу мастер теореме лако се закључује да је $T(n) = O(n \log n)$. Дакле, овај алгоритам је мање ефикасан од претходна два, али је и даље прилично употребљив, јер је много бољи од почетна два веома наивна покушаја.

```
int maksZbirSegmenta(const vector<int>& a, int l, int d) {
  if (l > d)
```

```
    return 0;
  int s = l + (d - l) / 2;
  int maks_zbir_levo = maksZbirSegmenta(a, l, s-1);
  int maks_zbir_desno = maksZbirSegmenta(a, s+1, d);
  int zbir_sredina = a[s];
  int maks_zbir_sredina = zbir_sredina;
  for (int i = s-1; i >= l; i--) {
    zbir_sredina += a[i];
    if (zbir_sredina > maks_zbir_sredina)
      maks_zbir_sredina = zbir_sredina;
  }
  zbir_sredina = maks_zbir_sredina;
  for (int i = s+1; i <= d; i++) {
    zbir_sredina += a[i];
    if (zbir_sredina > maks_zbir_sredina)
      maks_zbir_sredina = zbir_sredina;
  }
  return max({maks_zbir_levo, maks_zbir_desno, maks_zbir_sredina});
}

int maksZbirSegmenta(const vector<int>& a) {
  return maksZbirSegmenta(a, 0, a.size() - 1);
}
```

Ојачање индуктивне хипотезе

На идеји декомпозиције можемо изградити и ефикаснији алгоритам. Кључни увид је да се највећи збир сегмента око средњег елемента може добити као збир највећег суфикса низа лево од тог елемента и највећег префикса низа десно од тог елемента. Можемо ојачати индуктивну хипотезу и уместо да префикс и суфикс рачунамо у петљи, у линеарном времену, можемо претпоставити да за обе половине низа префикс и суфикс добијамо као резултат рекурзивног позива. То нам је довољно да одредимо максимални збир функције, али морамо вратити дуг и наша функција сада поред максималног збира сегмента мора израчунати и максимални збир префикса и максимални збир суфикса целог низа. Максимални збир префикса целог низа је већи број од максималног збира префикса левог дела и од збира целог левог дела и максималног збира префикса десног дела. Слично, максимални збир суфикса целог низа је већи од максималног збира суфикса десног дела и од збира максималног збира суфикса левог дела и целог десног дела. Зато је неопходно додатно ојачати индуктивну хипотезу и током рекурзије рачунати и збир целог низа.

Једначина која описује ову рекурзију је $T(n) = 2T(n/2) + O(1)$, па је сложеност овог решења линеарна тј. $O(n)$.

```
void maksZbirSegmenta(const vector<int>& a, int l, int d,
                      int& zbir, int& maks_zbir,
                      int& maks_prefiks, int& maks_sufiks) {
  if (l == d) {
    zbir = maks_zbir = maks_prefiks = maks_sufiks = a[l];
    return;
  }
  int s = l + (d - l) / 2;
  int zbir_levo, maks_zbir_levo, maks_sufiks_levo, maks_prefiks_levo;
  maksZbirSegmenta(a, l, s,
                   zbir_levo, maks_zbir_levo,
                   maks_prefiks_levo, maks_sufiks_levo);
  int zbir_desno, maks_zbir_desno, maks_sufiks_desno, maks_prefiks_desno;
  maksZbirSegmenta(a, s+1, d,
                   zbir_desno, maks_zbir_desno,
                   maks_prefiks_desno, maks_sufiks_desno);
  zbir = zbir_levo + zbir_desno;
  maks_prefiks = max(maks_prefiks_levo, zbir_levo + maks_prefiks_desno);
  maks_sufiks = max(maks_sufiks_desno, maks_sufiks_levo + zbir_desno);
```

```
  maks_zbir = max({maks_zbir_levo, maks_zbir_desno,
                   maks_sufiks_levo + maks_prefiks_desno});
}

int maksZbirSegmenta(const vector<int>& a) {
  int zbir, maks_zbir, maks_prefiks, maks_sufiks;
  maksZbirSegmenta(a, 0, a.size() - 1,
                   zbir, maks_zbir, maks_prefiks, maks_sufiks);
  return maks_zbir;
}
```

Види другачија решења овог задатка.

Задатак: Паметна куповина акција

Позната је цена акција током више дана. Напиши програм који одређује максималну зараду која се може остварити тако што се један дан акција купи и неки наредни дан прода. Ако су цене акција строго опадајуће, онда је зарада 0.

Улаз: Са стандардног улаза се уноси број n ($2 \leq n \leq 50000$), а затим у наредних n линија по један позитиван број који представља цену акција.

Излаз: На стандардни излаз исписати тражени износ максималне зараде.

Пример

Улаз	*Излаз*
7 3 5 8 4 2 6 9	7

Задатак: Најближи пар тачака

У датом скупу тачака у равни одредити колико је растојање између две тачке које су међусобно најближе.

Улаз: Са стандардног улаза се уноси број тачака n ($1 \leq n \leq 50000$), а затим у наредних n редова координате тачака (два цела броја између -10^9 и 10^9, раздвојена размаком).

Излаз: На стандардни излаз исписати тражено растојање, заокружено на пет децимала.

Пример

Улаз	*Излаз*
5 0 0 0 2 2 0 2 2 1 1	1.41421

Решење

Груба сила

Решење грубом силом подразумева испитивање свих парова тачака и сложеност му је $O(n^2)$.

```
#include <iostream>
#include <iomanip>
#include <algorithm>
```

```cpp
#include <vector>
#include <cmath>

using namespace std;

struct Tacka {
  int x, y;
};

double rastojanje(const Tacka& t1, const Tacka& t2) {
  double dx = t1.x - t2.x;
  double dy = t1.y - t2.y;
  return sqrt(dx*dx + dy*dy);
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  vector<Tacka> tacke(n);
  for (int i = 0; i < n; i++)
    cin >> tacke[i].x >> tacke[i].y;

  double d = numeric_limits<double>::max();
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++) {
      double dij = rastojanje(tacke[i], tacke[j]);
      if (dij < d)
        d = dij;
    }

  cout << fixed << showpoint << setprecision(5) << d << endl;

  return 0;
}
```

Декомпозиција

Један начин да се до решења дође ефикасније је да се примени декомпозиција.

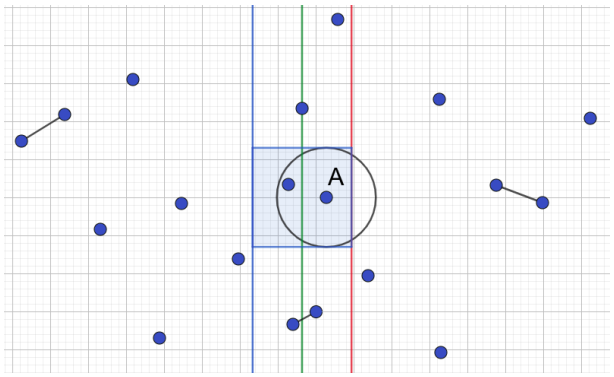
Базни случај представља ситуација у којој имамо мање од четири тачке, јер њих не можемо поделити у две половине у којима постоји бар по један пар тачака (а ако у скупу немамо бар 2 тачке, најмање растојање није јасно дефинисано). У том случају решење налазимо поређењем растојања свих парова тачака (пошто је тачака мало, овај корак је сложености $O(1)$).

Скуп тачака можемо једном вертикалном линијом поделити на две отприлике истобројне половине. Ако тачке сортирамо по координати x, вертикална линија може одговарати координати средишње тачке. Рекурзивно одређујемо најмање растојање у првој половини (то су тачке лево од вертикалне линије) и у другој половини (то су тачке десно од вертикалне линије). Најближи пар је такав да су (1) обе тачке у левој половини, (2) обе тачке у десној половини или (3) једна тачка је у левој, а друга у десној половини. За прва два случаја већ знамо решења и остаје да се размотри само трећи.

Нека је d_l минимално растојање тачака у левој половини, d_r минимално растојање тачака у десној половини, а d мање од та два растојања. Ако вертикална линија има x-координату x, тада је могуће одбацити све тачке које су лево од $x-d$ и десно од $x+d$, јер је њихово растојање до најближе тачке из супротне половине сигурно веће од d. Потребно је испитати све преостале тачке, тј. све тачке из појаса $[x-d, x+d]$, проверити да ли међу њима постоји неки пар тачака чије је растојање строго мање од d и вредност d ажурирати на вредност најмањег растојања таквог пара тачака. Проблем је то што у најгорем случају њих може бити пуно (могуће је да се свих n тачака нађе у том појасу) и ако испитујемо све парове, долазимо у најгорем случају до око $n^2/4$ поређења (ако је пола тачака лево, а пола десно од линије поделе). Ипак, проверу је могуће организовати

тако да се провери само мали број парова тачака.

Једноставности ради ћемо претпоставити да истовремено разматрамо све тачке унутар појаса $[x-d, x+d]$, без обзира са које стране вертикалне линије се налазе (унапред знамо да је провера тачака које су са исте стране вертикалне линије поделе непотребна, али не може нарушити коректност, док год смо сигурни да се пореде и сви потребни парови тачака са различите стране те линије). Сваку тачку A из појаса је довољно упоредити са оним тачкама које леже унутар круга са центром у тачки A и полупречником d, што омогућава значајна одсецања. Припадност кругу није једноставно проверити и зато уместо њега можемо разматрати квадрат странице дужине $2d$ на чијој се хоризонталној средњој линији налази тачка A. Тиме ће одсецање бити за нијансу мање него у случају круга, али ће детектовање тачака које припадају том правоугаонику бити веома једноставно. То ће бити све оне тачке из појаса којима је координата у у интервалу $[y_A - d, y_A + d]$.

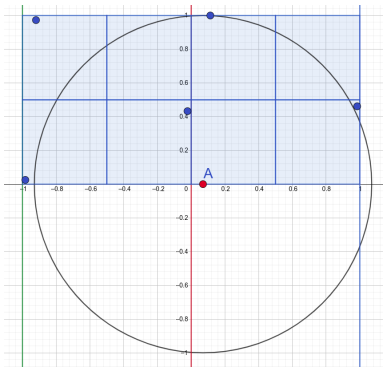


Слика 6.1: Најближи пар тачака у левом појасу, десном појасу и између појасева. Круг и квадрат којим су одређени кандидати око тачке A.

Даље смањење броја поређења можемо добити ако приметимо да сваки пар обрађујемо два пута (једном док обрађујемо тачке у околини прве, а једном док обрађујемо тачке у околини друге тачке). Можемо једноставно закључити да је довољно сваку тачку поредити само са оним тачкама које се налазе на истој висини као она или изнад ње. Дакле, сваку тачку је потребно упоредити само са тачкама чије x координате леже унутар интервала $[x-d, x+d]$ и чије у координате леже унутар интервала $[y_A, y_A+d]$. Први услов можемо обезбедити тако што пре поређења све тачке из појаса ширине d око вертикалне линије поделе издвојимо у посебан низ (за то нам је потребно $O(n)$ додатне меморије и времена). Други услов ефикасније можемо обезбедити ако све тачке тог помоћног низа сортирамо по координати у (за то нам је потребно време $O(y \log n)$ и затим тачке обрађујемо у неопадајућем редоследу y координата. За сваку тачку A обрађујемо само тачке које се налазе иза ње у сортираном низу и обрађујемо једну по једну тачку све док не наиђемо на тачку чија је координата у већа или једнака $y_A + d$ (она од тачке A не може бити на мањем растојању од d).

Одредимо сложеност претходног алгоритма. Алгоритам се састоји од два рекурзивна позива за двоструко мању димензију низа тачака и фазе добијања крајњег резултата на основу резултата рекурзивних позива и додатне анализе тачака у појасу $[x-d, x+d]$. Већ смо констатовали да издвајање тачака централног појаса захтева $O(n)$ меморије и времена и да сортирање тих тачака по координати у захтева додатних $O(n \log n)$ корака. Остаје још да се процени сложеност угнежђених петљи у којима се пореде тачке унутар појаса. Иако делује да је сложеност квадратна, елементарним геометријским резоновањем доказаћемо да је сложеност тог корака линеарна тј. $O(n)$ и да се у сваком кораку спољашње петље унутрашња петља може извршити само веома мали број пута (доказаћемо да је тај број извршавања ограничен одозго са 7, мада је у пракси он често и доста мањи од тога и за насумично генерисане тачке та петља се најчешће извршава 0, 1 или евентуално 2 пута).

За сваку тачку A можемо конструисати 8 квадрата димензије $d/2$, као што је приказано на слици (квадрати су уписани у појас $[x-d, x+d]$, у два реда од по четири квадрата и тачка A лежи на доњој ивици доњих квадрата).



Слика 6.2: Најближи пар тачака

Највеће растојање између две тачке унутар неког квадрата се постиже када они леже у његовим наспрамним теменима, а пошто је дужина дијагонале квадрата странице $\frac{d}{2}$ једнака $\frac{d\sqrt{2}}{2} \approx 0,70711 \cdot d$, растојање између сваке две тачке унутар истог квадрата је строго мање од d. Пошто сви квадрати леже било потпуно са леве стране вертикалне линије поделе, било са њене десне стране унутар сваког од квадрата се може наћи највише једна тачка нашег скупа (у супротном би се било са леве, било са десне стране централне линије поделе налазио пар тачака са растојањем строго мањим од d, што је контрадикторно са дефиницијом величине d). То значи да се изнад тачке A може налазити највише 7 тачака које припадају осталим квадратима (сама тачка A већ припада једном од квадрата) и да се све остале тачке које су изнад A налазе и изнад наших квадрата, што значи да им је растојање од A сигурно веће од d (јер им је вертикално растојање веће од d) и њих није потребно разматрати.

Тачке које су са исте стране линије поделе као и тачка A можемо просто прескочити у телу унутрашње петље и тако уштедити на рачунању њиховог растојања од тачке A, али експерименти показују да та уштеда није осетна. Друга могућност за имплементацију је да не чувамо све тачке из појаса у истом скупу, већ да их поделимо у два појаса и да затим да обрадимо прво све тачке из левог појаса гледајући растојања у односу на наредне највише 4 тачке из десног појаса, а затим да обрадимо све тачке из десног појаса гледајући растојања у односу на највише 4 тачке из левог појаса (јер у супротном појасу постоји 4 квадрата димезије $d/2$, за које смо доказали да не могу да садрже две тачке истовремено). Имплементација на тај начин је мало компликованија, а експерименти не указују на значајне добитке.

Дакле, након рекурзивних позива, за добијање коначног резултата је потребно извршити додатних $O(n \log n)$ корака и декомпозиција задовољава рекурентну једначину $T(n) = 2T(n/2) + O(n \log n)$. Решење ове једначине, на основу мастер теореме, је $O(n(\log n)^2)$.

```
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <vector>
#include <utility>
#include <limits>
#include <cmath>

using namespace std;

struct Tacka {
  int x, y;
};

double rastojanje(const Tacka& t1, const Tacka& t2) {
  double dx = t1.x - t2.x;
  double dy = t1.y - t2.y;
  return sqrt(dx*dx + dy*dy);
}

double najblizeTacke(vector<Tacka>& tacke, int l, int r, vector<Tacka>& pojas) {
```

```
  if (r - l + 1 < 4) {
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++) {
        double dij = rastojanje(tacke[i], tacke[j]);
        if (dij < d)
          d = dij;
      }
    return d;
  }

  int s = l + (r - l) / 2;
  double d1 = najblizeTacke(tacke, l, s, pojas);
  double d2 = najblizeTacke(tacke, s+1, r, pojas);
  double d = min(d1, d2);

  double dl = tacke[s].x - d, dr = tacke[s].x + d;

  int k = 0;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas[k++] = tacke[i];

  sort(begin(pojas), next(begin(pojas), k),
       [](const Tacka& t1, const Tacka& t2) {
         return t1.y < t2.y;
       });

  for (int i = 0; i < k; i++)
    for (int j = i+1; j < k && pojas[j].y - pojas[i].y < d; j++) {
      double dij = rastojanje(pojas[i], pojas[j]);
      if (dij < d)
        d = dij;
    }

  return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
  sort(begin(tacke), end(tacke),
       [](const Tacka& t1, const Tacka& t2) {
         return t1.x < t2.x;
       });
  vector<Tacka> pojas(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pojas);
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  vector<Tacka> tacke(n);
  for (int i = 0; i < n; i++)
    cin >> tacke[i].x >> tacke[i].y;
  cout << fixed << showpoint << setprecision(5)
       << najblizeTacke(tacke) << endl;
  return 0;
}
```

Сортирање обједињавањем

Сложеност се може поправити ако се сортирање по координати у врши истовремено са проналажењем најближег пара тачака, тј. ако се ојача индуктивна хипотеза и ако се претпостави да ће рекурзивни позив вратити растојање између најближе две тачке и уједно сортирати дате тачке по координати y. У кораку обједињавања два сортирана низа обједињујемо у један. То можемо урадити уобичајеним алгоритмом обједињавања, заснованом на техници два показивача, који обједињавање врши у линеарној сложености. У језику C++ тај алгоритам је доступан и помоћу библиотечке функције `merge`. На тај начин добијамо алгоритам који задовољава једначину $T(n) = 2T(n/2) + O(n)$ и сложености је $O(n \log n)$. Нагласимо да ова оптимизација није револуционарна, али може мало побољшати ефикасност.

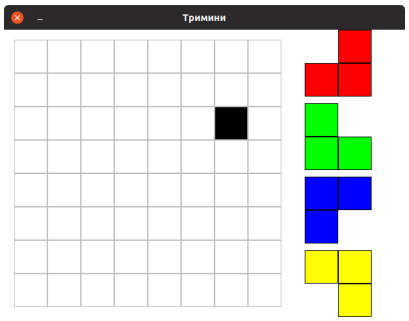
На нивоу имплементације, мало побољшање бисмо могли добити и тако што бисмо избегли алокације помоћног вектора унутар рекурзивних позива и код изменити тако да се у сваком рекурзивном позиву користи исти, унапред алоциран помоћни вектор. Још једна могућа оптимизација о којој би се могло размислити је смањивање броја операција кореновања.

```
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <vector>
#include <limits>
#include <cmath>
#include <utility>

using namespace std;

struct Tacka {
  int x, y;
};

bool porediX(const Tacka& t1, const Tacka& t2) {
  return t1.x <= t2.x;
}

bool porediY(const Tacka& t1, const Tacka& t2) {
  return t1.y <= t2.y;
}

double rastojanje(const Tacka& t1, const Tacka& t2) {
  double dx = t1.x - t2.x;
  double dy = t1.y - t2.y;
  return sqrt(dx*dx + dy*dy);
}

void ispisiTacke(vector<Tacka>& tacke, int l, int r) {
  for (int i = l; i <= r; i++)
    cout << "(" << tacke[i].x << ", " << tacke[i].y << ")" << " ";
  cout << endl;
}

double najblizeTacke(vector<Tacka>& tacke, int l, int r, vector<Tacka>& pojas) {
  if (r - l + 1 < 4) {
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++) {
        double dij = rastojanje(tacke[i], tacke[j]);
        if (dij < d)
          d = dij;
      }
```

```
    sort(next(begin(tacke), l), next(begin(tacke), r+1), porediY);
    return d;
  }

  int s = l + (r - l) / 2;
  int x = tacke[s].x;
  double d1 = najblizeTacke(tacke, l, s, pojas);
  double d2 = najblizeTacke(tacke, s+1, r, pojas);
  double d = min(d1, d2);

  double dl = x - d, dr = x + d;

  merge(next(begin(tacke), l), next(begin(tacke), s+1),
        next(begin(tacke), s+1), next(begin(tacke), r+1),
        begin(pojas), porediY);
  copy(begin(pojas), next(begin(pojas), r - l + 1), next(begin(tacke), l));

  int k = 0;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas[k++] = tacke[i];

  for (int i = 0; i < k; i++)
    for (int j = i+1; j < k && pojas[j].y - pojas[i].y < d; j++) {
      double dij = rastojanje(pojas[i], pojas[j]);
      if (dij < d)
        d = dij;
    }

  return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
  // Iz nekog cudnog razloga baguje:
  // sort(begin(tacke), end(tacke), porediX);
  stable_sort(begin(tacke), end(tacke), porediX);
  vector<Tacka> pojas(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pojas);
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  vector<Tacka> tacke(n);
  for (int i = 0; i < n; i++)
    cin >> tacke[i].x >> tacke[i].y;


  cout << fixed << showpoint << setprecision(5)
       << najblizeTacke(tacke) << endl;
  return 0;
}
```

Задатак: Тримини

Нека је дата табла димензије 8×8 на којој недостаје једно поље. Задатак је попунити преостала 63 поља триминима (облицима који се добију када се из квадрата димензије 2×2 избаци једно поље).



Слика 6.3: Тримини

Улаз: Са стандардног улаза се учитавају два цела броја између 0 и 7, раздвојена размаком која представљају координате (врсту и колону) поља које је избачено са табле.

Излаз: На стандардни излаз исписати матрицу карактера која представља таблу прекривену триминима. На месту недостајућег поља треба да буде исписан размак, а тримини се представљају различитим карактерима (на пример, малим словима енглеске абецеде). Решење није јединствено.

Пример

Улаз	*Излаз*

`3 5`	`cceemmoo`
	`cbbemllo`
	`dbffnnlp`
	`ddfan pp`
	`hhjaartt`
	`hgjjrrqt`
	`iggksqqu`
	`iikkssuu`

Задатак: Хоризонт

Силуета зграде задаје се са три цела позитивна броја, а то су редом лева координата, висина и десна координата.

На хоризонту се виде само силуете зграда, тако да се линија хоризонта састоји од наизменичних хоризонталних и вертикалних сегмената. Захваљујући томе, линија хоризонта може да се зада низом темена, при чему линију пратимо од темена до темена идући прво хоризонтално надесно, а затим вертикално навише или наниже.

Напиши прогам који учитава силуете зхрада, а исписује линију хоризонта.

Улаз: Са стандардног улаза се у првом реду налази број n ($1 \leq n \leq 50\,000$), а у наредних n редова по три цела позитивна броја раздвојена размаком, који описују једну зграду. Ови бројеви не прелазе милион.

Излаз: На стандардни излаз исписати координате темена која задају линију хоризонта. Темена се исписују слева надесно, свако теме у посебном реду, а координате сваког темена раздвојити размаком.

Пример 1

Улаз	*Излаз*
`3`	`2 10`
`2 10 8`	`5 14`
`4 7 6`	`9 0`
`5 14 9`	

Пример 2

Улаз	*Излаз*
`2`	`2 10`
`2 10 8`	`8 0`
`11 3 14`	`11 3`
	`14 0`

Задатак: Трећи обилазак

Постоје три уобичајена редоследа обилажења чворова бинарног стабла:

- КЛД - најпре корен, па лево подстабло, па десно подстабло

- ЛКД - најпре лево подстабло, па корен, па десно подстабло
- ЛДК - најпре лево подстабло, па десно подстабло, па корен

У сваком од обилазака се подразумеава да се редослед обиласка подстабла наслеђује, тј. да се користи исто правило као за цело стабло

Нека је сваки чвор стабла означен различитим малим словом енглеског алфабета. На тај начин се сваким од поменутих обилазака добија по једна ниска.

Напиши прогам који учитава ниске које одговарају обиласцима КЛД и ЛКД, а исписује ниску која одговара обиласку ЛДК.

Улаз: Са стандардног улаза се учитавају две ниске, свака у посебном реду. Ниске одговарају обиласцима КЛД и ЛКД, а састоје се од различитих малих слова енглеског алфабета (и немају више од по 26 слова).

Излаз: На стандардни излаз исписати ниску која одговара обиласку ЛДК.

Пример

Улаз	*Излаз*
`abecfg` `beafcg`	`ebfgca`

Решење

Основни случај је стабло без чворова, чијим се обиласком добија празна ниска.

На даље ћемо ниске звати по обиласцима из којих се добијају.

Нека сада стабло има бар један чвор. Прво слово ниске КЛД мора да означава корен. Пронађимо исто слово у ниски ЛКД. Број слова која претходе корену у ниски ЛКД одређује дужину левог подстабла, а број слова која следе одређује дужину десног подстабла. Када знамо дужине подстабала, лако можемо да одредимо делове ниске КЛД који одговарају КЛД записима левог и десног подстабла (свака подниска има онолико слова, колико одговарајуће подстабло има чворова).

Преостали део посла се своди на решавање истог типа задатка за два мања стабла, што можемо да урадимо рекурзивним позивима исте функције.

Да бисмо избегли формирање и копирање делова ниски, користићемо две глобалне ниске, које одговарају датим обиласцима целог стабла.

Уместо да рекурзивној функцији прослеђујемо ниске настале обиласцима неког подстабла, прослеђиваћемо позиције почетака одговарајућих подниски у двема глобалним нискама. Поред ових позиција, функцији прослеђујемо и величину (број чворова) подстабла.

```
#include <iostream>
#include <string>

using namespace std;
string kld, lkd;

string NadjiLDK(int kld_poc, int lkd_poc, int n) {
    if (n == 0) return "";

    char koren = kld[kld_poc];
    int m = lkd.find(koren);
    int n_levo = m - lkd_poc;
    int n_desno = n - n_levo - 1;
    return NadjiLDK(kld_poc+1, lkd_poc, n_levo) +
           NadjiLDK(kld_poc+1+n_levo, m+1, n_desno) +
           koren;
}

int main() {
  cin >> kld;
  cin >> lkd;
```

```
  int n = kld.size();
  cout << NadjiLDK(0, 0, n) << endl;
  return 0;
}
```

Глава 7

Динамичко програмирање

7.1 Бројање комбинаторних објеката

Задатак: Број комбинација

Напиши програм који одређује број комбинација без понављања дужине k из скупа од k елемената (тј. број различитих комбинација у игри лото ако се из бубња који садржи n лоптица извлачи њих k).

Улаз: Са стандардног улаза се извлачи број k ($1 \leq k \leq n$) и број n ($1 \leq n \leq 40$).

Излаз: На стандардни излаз исписати тражени број комбинација.

Пример 1

Улаз *Излаз*
```
3          10
5
```
Објашњење

То су комбинације $(1, 2, 3)$, $(1, 2, 4)$, $(1, 2, 5)$ $(1, 3, 4)$, $(1, 3, 5)$, $(1, 4, 5)$, $(2, 3, 4)$, $(2, 3, 5)$, $(2, 4, 5)$ и $(3, 4, 5)$.

Пример 2

Улаз

```
7
39
```

Излаз

```
15380937
```

Решење

Иако постоје разни начини да се до решења овог задатка дође, приказаћемо технику засновану на томе да програм који набраја све комбинаторне објекте мало по мало трансформишемо до ефикасног прорама који их броји. Ова техника није специфична за комбинације без понављања и може се применити на бројање било које врсте комбинаторних објеката које набрајамо рекурзивном функцијом.

Рецимо да је број комбинација једнак биномном коефицијенту

$$\binom{n}{k} = \frac{n!}{(n-k)!k!},$$

међутим израчунавање на основу директне примене ове формуле би веома брзо довело до прекорачења (услед веома брзог раста факторијелске функције). Мало боља ситуација је да се израчуна

$$\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!},$$

но ни то у потпуности не уклаља проблем прекорачења, јер именилац може бити превелики.

Можемо кренути од процедуре коју смо извели за генерисање свих комбинација у задатку Све комбинације, у којој се одржава интервал $[n_{min}, n_{max}]$ из којег се могу узети вредности којима се проширује започета комбинација и у којој се кроз два рекурзивна позива разматра могућност да се вредност n_{min} уврсти у комбинацију и могућност да се не уврсти.

Уместо процедуре која исписује комбинације, дефинишемо фукцију која враћа број комбинација. Ако нам је битан број комбинација, а не и саме комбинације, тада можемо у потпуности избацити из игре низ који се попуњава и уместо њега прослеђивати само његову дужину k.

```
#include <iostream>
#include <vector>

using namespace std;

long long brojKombinacija(int i, int k,
                          int n_min, int n_max) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (i == k) return 1;
  // ako niz nije moguće popuniti do kraja, tada nema kombinacija
  if (k - i > n_max - n_min + 1)
    return 0;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(i+1, k, n_min+1, n_max) +
         brojKombinacija(i, k, n_min+1, n_max);
}

long long brojKombinacija(int K, int N) {
  return brojKombinacija(0, K, 1, N);
}

int main() {
  int K, N;
  cin >> K >> N;
  cout << brojKombinacija(K, N) << endl;
  return 0;
}
```

Можемо приметити да нам конкретне вредности k и i нису битне, већ је битан само број елемената у интервалу $[i, k)$ тј. разлика $k - i$. Слично, нису нам битне ни конкретне вредности n_{max} и n_{min} већ само број елемената у сегменту $[n_{min}, n_{max}]$ тј. вредност $n_{max} - n_{max} + 1$. Ако те две величине заменимо са k тј. n добијамо наредну дефиницију.

```
#include <iostream>
#include <vector>

using namespace std;

long long brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako niz nije moguće popuniti do kraja, tada nema kombinacija
  if (k > n) return 0;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}


int main() {
  int K, N;
```

```
  cin >> K >> N;
  cout << brojKombinacija(K, N) << endl;
  return 0;
}
```

Ако функцију позовемо за вредности $k \leq n$, случај $k > n$ може наступити једино из другог рекурзивног позива за $k = n$ (јер однос између k и n у првом рекурзивном позиву остаје непромењен, а у другом се мења само за 1). Међутим, у случају позива функције за $k = n$ добиће се увек повратна вредност 1 (други рекурзивни позив ће увек враћати нуле, а први ће проузроковати смањивање оба аргумента све док се не дође до $k = n = 0$, када ће се 1 вратити на основу првог излаза из рекурзије), што је сасвим у складу са тим да тада постоји само једна комбинација. На основу овога из рекурзије можемо изаћи за $k = n$ вративши вредност 1, чиме онда елиминишемо потребу за провером да ли је $k > n$ (наравно, под претпоставком да ћемо функцију позивати само за $k \leq n$).

Примећујемо да смо овом трансформацијом добили чувене особине биномних коефицијената.

$$\binom{n}{0} = 1, \qquad \binom{n}{n} = 1, \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Оне чине основу Паскаловог троугла у ком се налазе биномни коефицијенти.

```
1                      (0,0)
1  1                   (1,0) (1,1)
1  2  1                (2,0) (2,1) (2,2)
1  3  3  1             (3,0) (3,1) (3,2) (3,3)
1  4  6  4  1          (4,0) (4,1) (4,2) (4,3) (4,4)
1  5 10 10  5  1       (5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
1  6 15 20 15  6  1    (6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6)
```

Прва веза говори да су елементи прве колоне увек једнаки 1, друга да су на крају сваке врсте елементи такође једнаки 1, а трећа да је сваки елемент у троуглу једнак збиру елемента непосредно изнад њега и елемента непосредно испред тог.

```
#include <iostream>
#include <vector>

using namespace std;

long long brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako treba popuniti još tačno n elemenata, tada postoji
  // tačno jedna kombinacija
  if (k == n) return 1;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}

int main() {
  int K, N;
  cin >> K >> N;
  cout << brojKombinacija(K, N) << endl;
  return 0;
}
```

Иако коректна, горња функција је неефикасна и може се поправити техником динамичког програмирања. Најједноставније прилагођавање је да се употреби мемоизација. Пошто функција има два параметра, за мемоизацију ћемо употребити матрицу. Ако се $\binom{n}{k}$ памти у матрици на позицији (n, k), матрицу можемо алоцирати на $n + 1$ врста, где последња врста има $n + 1$ елемената, а свака претходна један елемент мање (у матрицу ће се попуњавати елементи Паскаловог троугла). Пошто нас неће занимати вредности веће од

полазног k и пошто се и k и n смањују током рекурзије, при чему је $k \leq n$, можемо и одсећи део троугла десно од позиције k.

Пошто су бројеви комбинација увек већи од нуле, вредности 0 у матрици ће нам означавати да позив за те параметре још није извршен.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

long long brojKombinacija(int k, int n, vector<vector<long long>>& memo) {
  // ako smo već računali broj kombinacija, ne računamo ga ponovo
  if (memo[n][k] != 0) return memo[n][k];

  // broj kombinacija na početku i na kraju svake vrste jednak je 1
  if (k == 0 || k == n) return memo[n][k] = 1;
  // broj kombinacija u sredini jednak je zbiru
  // broja kombinacija iznad i iznad levo od tekućeg elementa
  return memo[n][k] = brojKombinacija(k-1, n-1, memo) +
                      brojKombinacija(k, n-1, memo);
}

long long brojKombinacija(int K, int N) {
  // alociramo prostor za rezultate rekurzivnih poziva koji se
  // mogu desiti i popunjavamo matricu nulama
  vector<vector<long long>> memo(N+1);
  for (int n = 0; n <= N; n++)
    memo[n].resize(min(K+1, n+1), 0);
  // pozivamo funkciju koja će izračunati traženi broj
  return brojKombinacija(K, N, memo);
}

int main() {
  int K, N;
  cin >> K >> N;
  cout << brojKombinacija(K, N) << endl;
  return 0;
}
```

Уместо мемоизације можемо употребити и динамичко програмирање навише, ослободити се рекурзије и попунити троугао врсту по врсту наниже. Попуњавање целог троугла је прилично једноставно.

```
#include <iostream>
#include <vector>

using namespace std;

long long brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje celog trougla
  vector<vector<long long>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(n+1);
  // obrađujemo vrstu po vrstu
  for (int n = 0; n <= N; n++) {
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata iznad njih
    for (int k = 1; k < n; k++)
```

```
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // na kraju svake vrste nalazi se 1
    dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}

int main() {
  int K, N;
  cin >> K >> N;
  cout << brojKombinacija(K, N) << endl;
  return 0;
}
```

И у овом случају можемо одсећи непотребне десне колоне у троуглу.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

long long brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje relevantnog dela trougla
  vector<vector<long long>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(min(K+1, n+1));
  // trougao popunjavamo kolonu po kolonu
  for (int n = 0; n <= N; n++) {
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata iznad njih
    for (int k = 1; k <= min(n-1, K); k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // ako je potrebno da znamo krajnji element kolone, postavljamo
    // ga na vrednost 1
    if (n <= K)
       dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}

int main() {
  int K, N;
  cin >> K >> N;
  cout << brojKombinacija(K, N) << endl;
  return 0;
}
```

Пажљивијом анализом претходног кода видимо да, како је то обично случај у динамичком програмирању, не морамо истовремено чувати све елементе матрице, јер свака врста зависи само од претходне и довољно је уместо матрице чувати само њене две врсте (претходну и текућу). Заправо, довољно је чувати само један вектор врсту ако је пажљиво попуњавамо и ако током њеног ажурирања у једном њеном делу чувамо текућу, а у другом наредну врсту. Пошто елемент (n, k) зависи од елемента $(n-1, k-1)$ и од елемента $(n-1, k)$ значи да сваки елемент зависи од елемената који су лево од њега, али не од елемената који су десно од њега. Зато ћемо вектор попуњавати здесна налево. Претпоставићемо да током ажурирања важи инваријанта да се на позицијама строго већим од k налазе елементи врсте n, а да се на позицијама мањим или једнаким од k налазе елементи врсте $n-1$. Ажурирање започиње тиме што на крај врсте допишемо вредност 1 (осим у

случају када вршимо сасецање десног дела троугла) и наставља се тако што се елемент на позицији k увећа за вредност на позицији $k-1$. Заиста, пре ажурирања се на позицији k налази вредост троугла са позиције $(n-1,k)$, док се на позицији $k-1$ налази вредност троугла са позиције $(n-1,k-1)$. Њихов збир је вредност троугла на позицији (n,k), па се он уписује на позицију k и након тога се k смањује за 1, чиме се инваријанта одржава. Ажурирање се врши до позиције $k=1$, јер се на позицији $k=0$ у свим врстама налази вредност 1.

Меморијска сложеност овог решења је $O(k)$, док је временска $O(n \cdot k)$. Приметимо како смо од веома неефикасног решења експоненцијалне сложености техником динамичког програмирања добили веома ефикасно и уз то прилично једноставно решење.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

long long brojKombinacija(int K, int N) {
  // tekuća vrsta
  vector<long long> dp(K+1);
  // na početku svake vrste nalazi se 1
  dp[0] = 1;
  // trougao popunjavamo vrstu po vrstu
  for (int n = 1; n <= N; n++) {
    // vrstu ažuriramo zdesna nalevo
    // na kraju svake vrste nalazi se 1
    if (n <= K) dp[n] = 1;
    // ažuriramo unutrašnje elemente
    for (int k = min(n-1, K); k > 0; k--)
        dp[k] += dp[k-1];
  }
  // vraćamo traženi rezultat
  return dp[K];
}


int main() {
  int K, N;
  cin >> K >> N;
  cout << brojKombinacija(K, N) << endl;
  return 0;
}
```

Рецимо и да смо могли кренути од алгоритма набрајања свих комбинација у ком се у петљи разматрају сви кандидати за елемент на текућој позицији. Тиме би се добио алгоритам који би елемент $\binom{n}{k}$ рачунао по следећој формули:

$$\binom{n}{k} = \sum_{n'=k}^{n} \binom{n'}{k-1}.$$

Задатак: Број комбинација са понављањем

Напиши програм који одређује на колико се начина може извући k лоптица из бубња који садржи n различитих лоптица, ако се свака извучена лоптица враћа у бубањ пре новог извлачења.

Улаз: Са стандардног улаза се учитава број k ($1 \leq k \leq n$) и n ($1 \leq n \leq 30$), сваки из посебног реда.

Излаз: На стандардни излаз исписати тражени број комбинација са понављањем.

Пример

Улаз	*Излаз*
2	6
3	

Објашњење

То су комбинације $(1,1)$, $(1,2)$, $(1,3)$, $(2,2)$, $(2,3)$ и $(3,3)$.

Решење

Ако је потребно изабрати 0 елемената из скупа од n елемената то је могуће урадити само на један начин. Ако је потребно изабрати $k > 0$ елемената из празног скупа, то није могуће учинити. У супротном, све комбинације можемо поделити на оне које почињу најмањим елементом скупа од n елемената и на оне који не почињу њиме. Можемо дакле, узети најмањи елемент скупа и затим гледати све могуће начине да се одабере $k-1$ елемент из истог n-точланог скупа или можемо свих k елемената изабрати из скупа из ког је избачен тај најмањи елемент. Овим добијамо веома једноставну рекуренту везу на основу које можемо имплементирати рекурзивну функцију (која ће, додуше, бити неефикасна).

$$
\begin{aligned}
&f(0,n) = 1 \\
&f(k,0) = 0, \quad \text{за } k > 0 \\
&f(k,n) = f(k-1,n) + f(k,n-1), \quad \text{за } k,n > 0
\end{aligned}
$$

```
#include <iostream>

using namespace std;

int brojKombinacijaSaPonavljanjem(int k, int n) {
  if (k == 0) return 1;
  if (n == 0) return 0;
  return brojKombinacijaSaPonavljanjem(k-1, n) +
         brojKombinacijaSaPonavljanjem(k, n-1);
}

int main() {
  int K, N;
  cin >> K >> N;
  cout << brojKombinacijaSaPonavljanjem(K, N) << endl;
  return 0;
}
```

Неефикасност претходне функције потиче од тога што се идентични рекурзивни позиви понављају више пута. То је могуће поправити техником динамичког програмирања. Број комбинација за разне вредности (n,k) се може распоредити у правоугаоник.

```
                          (k,n)
1  1   1   1   1   1      (0,0) (0,1) (0,2) (0,3) (0,4) (0,5)
0  1   2   3   4   5      (1,0) (1,1) (1,2) (1,3) (1,4) (1,5)
0  1   3   6  10  15      (2,0) (2,1) (2,2) (2,3) (2,4) (2,5)
0  1   4  10  20  35      (3,0) (3,1) (3,2) (3,3) (3,4) (3,5)
0  1   5  15  35  70      (4,0) (4,1) (4,2) (4,3) (4,4) (4,5)
0  1   6  21  56 126      (5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
```

Претходна рекурентна веза нам говори да се у првој колони налазе нуле, у првој врсти јединице, а да је сваки елемент у остатку матрице једнак збиру елемената непосредно изнад и лево од њега. Матрицу можемо попунити врсту по врсту, но, можемо направити и наредну меморијску оптимизацију.

Елементи сваке врсте зависе само од елемената те и претходне врсте, па није неопходно чувати целу матрицу, већ је довољно чувати само један низ, који ће током ажурирања чувати неке елементе претходне и неке елементе текуће врсте. Врста за следеће k се може добити ажурирањем врсте за претходно k. Пошто сваки елемент у правоугаонику зависи од вредности изнад и лево од себе, врсту можемо ажурирати слева надесно. Инваријанта је да се у тренутку ажурирања позиције n, на позицијама строго мањим од n налазе вредности

из текуће врсте k, а на позицијама од n надаље се налазе вредности из претходне врсте $k-1$. Елемент на позицији n који садржи вредност са позиције $(k-1, n)$ правоугаоника увећавамо за вредност лево од њега који садржи вредност $(k, n-1)$ и тако добијамо вредност елемента на позицији (k, n). Увећавањем вредности n за 1 се одржава инваријанта.

```
#include <iostream>
#include <vector>

using namespace std;

long long brojKombinacijaSaPonavljanjem(int K, int N) {
  vector<long long> dp(N+1, 1);
  for (int k = 1; k <= K; k++) {
    dp[0] = 0;
    for (int n = 1; n <= N; n++)
      dp[n] += dp[n-1];
  }
  return dp[N];
}

int main() {
  int K, N;
  cin >> K >> N;
  cout << brojKombinacijaSaPonavljanjem(K, N) << endl;
  return 0;
}
```

Постоји и веома згодан алгоритам да се комбинације са понављањем сведу на обичне комбинације (чији смо број рачунали у задатку Број комбинација). Замислимо да смо поређали све елементе од 1 до n и да правимо програм за робота који ће одабрати комбинацију тако што креће од првог броја и у сваком тренутку или може да узме тај број (операција $+$) или да се помери на следећи број (операција $\rightarrow$), све док не стигне до последњег броја, при чему треба укупно да узме k бројева. На пример, ако је низ бројева 1, 2, 3, 4 и бира се 4 броја, онда програм $\rightarrow, +, +, \rightarrow, \rightarrow, +, +$ означава комбинацију $2, 2, 4, 4$. Питање, дакле, сводимо на то како распоредити $n-1$ стрелица и k плусева, што одговара питању како распоредити k плусева на $n+k-1$ позиција, тако да је је укупан број комбинација са понављањем једнак $\binom{n+k-1}{k}$.

```
#include <iostream>
#include <vector>

using namespace std;

long long brojKombinacija(int K, int N) {
  // tekuća vrsta
  vector<long long> dp(K+1, 0);
  // na početku svake vrste nalazi se 1
  dp[0] = 1;
  // trougao popunjavamo kolonu po kolonu
  for (int n = 1; n <= N; n++) {
    // vrstu ažuriramo zdesna nalevo
    // na kraju svake vrste nalazi se 1
    if (n <= K) dp[n] = 1;
    // ažuriramo unutrašnje elemente
    for (int k = min(n-1, K); k > 0; k--)
      dp[k] += dp[k-1];
  }
  // vraćamo traženi rezultat
  return dp[K];
}
```

```
long long brojKombinacijaSaPonavljanjem(int K, int N) {
  return brojKombinacija(K, N + K - 1);
}

int main() {
  int K, N;
  cin >> K >> N;
  cout << brojKombinacijaSaPonavljanjem(K, N) << endl;
  return 0;
}
```

Задатак: Број партиција

Партиција позитивног природног броја n је растављање броја n на збир неколико позитивних природних бројева при чему је редослед сабирака небитан (стога можемо претпоставити да је тај редослед или увек нерастући или увек неопадајући). На пример, ако је редослед нерастући, партиције броја 4 су $1+1+1+1$, $2+1+1$, $2+2$, $3+1$, 4. Написати програм који одређује број партиција за дати природан број n.

Улаз: Прва и једина линија стандардног улаза садржи природан број n ($n \leq 100$).

Излаз: На стандардном излазу приказати у првој линији број партиција природног броја n.

Пример 1

Улаз *Излаз*

```
6          11
```

Објашњење

Ако су партиције са неопадајуће сортираним сабирцима, то су партиције:

```
1+1+1+1+1+1
1+1+1+1+2
1+1+1+3
1+1+2+2
1+1+4
1+2+3
1+5
2+2+2
2+4
3+3
6
```

Пример 2

Улаз

```
100
```

Излаз

```
190569292
```

Решење

Рекурзивне процедуре које набрајају све партиције, које су описане у задатку Све партиције се могу прилагодити тако да израчунају број партиција.

Свака партиција има свој први сабирак. Свакој партицији броја n којој је први сабирак s (при чему је $1 \leq s \leq n$) једнозначно одговара нека партиција броја $n - s$. Наметнућемо услов да су сабирци у свакој партицији сортирани нерастуће. Зато, ако је први сабирак s, сви сабирци иза њега морају да буду мањи или једнаки од s. Зато нам није довољно само да умемо да пребројимо све партиције броја $n - s$, већ је потребно да ојачамо индуктивну хипотезу. Означимо са $p_{n,s_{max}}$ број партиција броја n у којима су сви сабирци мањи или једнаки од s_{max}.

- Базу индукције чини случај $n = 0$, јер број нула има само једну партицију која не садржи сабирке. Дакле, важи да је $p_{0,s_{max}} = 1$. Ако је n веће од нула и $s_{max} = 0$, тада не постоји ни једна партиција,

јер од сабирака који су сви једнаки нули (јер сви морају да буду мањи или једнаки s_{max}) не можемо никако направити неки позитиван број. Дакле, за $n > 0$ важи да је $p_{n,0} = 0$.

- Индуктивни корак можемо остварити на више начина. Најједноставнији је следећи. Приликом израчунавања $p_{n,s_{max}}$ можемо разматрати два случаја: да се у збиру не јавља сабирак s_{max} или да се у збиру јавља сабирак s_{max}. Ако се у збиру не јавља сабирак s_{max}, тада је највећи сабирак $s_{max} - 1$ и број таквих партиција је $p_{n,s_{max}-1}$. Други случај је могућ само када је $n \geq s_{max}$ и број таквих партиција је $p_{n-s_{max},s_{max}}$. На основу овога, добијамо наредну рекурзивну функцију.

```
#include <iostream>
using namespace std;

// particije broja n u kojima je najveci sabirak jednak smax
int brojParticija(int n, int smax) {
  if (n == 0) return 1;
  if (smax == 0) return 0;
  int broj = brojParticija(n, smax - 1);
  if (n >= smax)
    broj += brojParticija(n - smax, smax);
  return broj;
}

int brojParticija(int n) {
  // particije broja n u kojima je najveći sabirak jednak n
  return brojParticija(n, n);
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
}
```

Рекурзивну функцију за израчунавање броја партиција можемо добити и тако што на текуће место у партицији постављамо све могуће кандидате за проширивање текуће партиције (то су сви бројеви s од 1 до мањег од бројева s_{max} и n) и настављамо генерисање партиција броја $n - s$ код којих је највећи сабирак једнак s.

```
#include <iostream>
#include <algorithm>
using namespace std;


// particije broja n u kojima je najveći sabirak jednak smax
int brojParticija(int n, int smax) {
   if (n == 0) return 1;
   if (smax == 0) return 0;
   int broj = 0;
   for (int s = min(smax, n); s >= 1; s--)
     broj += brojParticija(n - s, s);
   return broj;
}

int brojParticija(int n) {
  // particije broja n u kojima je najveći sabirak jednak n
  return brojParticija(n, n);
}

int main() {
  int n;
  cin >> n;
```

```
  cout << brojParticija(n) << endl;
}
```

Ако уместо услова да су сабирци уређени нерастуће ставимо услов да су сабирци уређени неопадајуће, тада уз број чије партиције тражимо шаљемо и број s_{min} који представља доњу границу наредних сабирака у партицији (сви сабирци морају да имају вредност бар s_{min}).

- Базу чини случај када је $n = 0$, јер број нула има само једну партицију која не садржи сабирке. Такође, када је $s_{min} > n$, тада је број партиција нула, јер ниједна партиција не може да има сабирак већи од збира.
- Број партиција броја n које садрже s_{min} једнак је броју партиција вредности $n - s_{min}$ са минималним сабирком s_{min}, док је број партиција које не садрже s_{min} једнак броју партиција броја n у којима је најмањи сабирак $s_{min} + 1$. Дакле, $p_{n,s_{min}} = p_{n-s_{min},s_{min}} + p_{n,s_{min}+1}$.

```
#include <iostream>
using namespace std;

// particije broja n u kojima je najveci sabirak jednak smax
int brojParticija(int n, int smin) {
  if (n == 0) return 1;
  if (smin > n) return 0;
  return  brojParticija(n, smin + 1) +
    brojParticija(n - smin, smin);
}

int brojParticija(int n) {
  // particije broja n u kojima je najmanji sabirak jednak 1
  return brojParticija(n, 1);
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
}
```

Можемо формулисати још једно решење у коме се броје неопадајуће сортиране партиције у којем у петљи разматрамо све могућности за вредност наредног сабирка. Размотримо, на пример, партиције броја 6.

```
1 1 1 1 1 1
1 1 1 1 2
1 1 1 3
1 1 2 2
1 1 4
1 2 3
1 5
2 2 2
2 4
3 3
6
```

Приметимо да је максимална вредност првог сабирка у свим партицијама осим у оној једночланој једнакој броју n, мања или једнака $\frac{n}{2}$ (у претходном примеру ни једна партиција не почиње ни са 4, ни са 5). Заиста, ако би први сабирак био строго већи од $\frac{n}{2}$ и ако партиција не би била једночлана и други сабирак би морао бити строго већи од $\frac{n}{2}$, па би збир био строго већи од n, што је немоуће. Стога једночлану партицију посебно урачунавамо, а рекурзивно генеришемо партиције за све могуће вредности првог сабирка од s_{min} до $\lfloor \frac{n}{2} \rfloor$.

```
#include <iostream>

using namespace std;
```

```
int brojParticija(int n, int smin) {
  if (n == 0)
    return 0;
  int br = 1;
  for(int i = smin; i <= n/2; i++)
    br += brojParticija(n - i, i);
  return br;
}

int brojParticija(int n) {
  return brojParticija(n, 1);
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
  return 0;
}
```

Сва решења заснована на простој рекурзији су неефикасна, јер долази до понављања идентичних рекурзивних позива и могу се поправити динамичким програмирањем. Прикажимо то на примеру бројања нерастуће уређених пермутација у којима вршимо два рекурзивна позива.

Кренимо са мемоизацијом. Уводимо матрицу димензије $(n+1) \times (n+1)$ коју попуњавамо вредностима -1, чиме означавамо да резултат позива функције још није познат. Пре него што кренемо са израчунавањем проверавамо да ли је у матрици вредност различита од -1 и ако јесте, враћамо ту упамћену вредност. Пре сваке повратне вредности функције резултат памтимо у матрици.

```
#include <iostream>
#include <vector>
using namespace std;

int brojParticija(int n, int smax, vector<vector<int>>& memo) {
  if (memo[n][smax] != -1)
    return memo[n][smax];
  if (n == 0) return memo[n][smax] = 1;
  if (smax == 0) return memo[n][smax] = 0;
  int broj = brojParticija(n, smax-1, memo);
  if (n >= smax)
    broj += brojParticija(n-smax, smax, memo);
  return memo[n][smax] = broj;
}

int brojParticija(int n) {
  vector<vector<int>> memo(n + 1);
  for (int i = 0; i <= n; i++)
    memo[i].resize(n+1, -1);
  return brojParticija(n, n, memo);
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
}
```

Уместо мемоизације можемо употребити и динамичко програмирање навише. Прикажимо табелу вредности функције за $n = 7$.

```
n\smax   0  1  2  3  4  5  6  7
```

```
0        1  1  1  1  1  1  1  1
1        0  1  1  1  1  1  1  1
2        0  1  2  2  2  2  2  2
3        0  1  2  3  3  3  3  3
4        0  1  3  4  5  5  5  5
5        0  1  3  5  6  7  7  7
6        0  1  4  7  9 10 11 11
7        0  1  4  8 11 13 14 15
```

На основу базе индукције знамо да ће сви елементи прве врсте бити једнаки 1, а да ће у првој колони сви елементи осим почетног бити једнаки 0. Један од начина да се матрица попуњава је постепено увећавајући вредност n, тј. попуњавајући врсту по врсту.

Елемент $p_{n,s}$ зависи од елемената $p_{n,s-1}$ и (ако је $n \geq s$) $p_{n-s,s}$ и ако се врсте попуњавају од горе наниже и слева надесно, приликом његовог израчунавања оба елемента од којих зависи су већ израчуната, што даје коректан алгоритам.

```
#include <iostream>
#include <vector>

using namespace std;

int brojParticija(int N) {
  // alociramo matricu
  vector<vector<int>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(N+1);
  // popunjavamo prvu vrstu
  for (int smax = 0; smax <= N; smax++)
    dp[0][smax] = 1;
  // popunjavamo preostale elemente prve kolone
  for (int n = 1; n <= N; n++)
    dp[n][0] = 0;
  // popunjavamo jednu po jednu vrstu
  for (int n = 1; n <= N; n++)
    for (int smax = 1; smax <= N; smax++) {
      dp[n][smax] = dp[n][smax-1];
      if (n >= smax)
        dp[n][smax] += dp[n-smax][smax];
    }
  return dp[N][N];
}


int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
  return 0;
}
```

И временска и меморијска сложеност претходног алгоритма је $O(n^2)$ иако се матрица попуњава врсту по врсту, то није могуће поправити (јер елементи зависе од елемената који се јављају не само у претходној, већ и у ранијим врстама, тако да је потребно да истовремено чувамо све претходне врсте). Међутим, ако матрицу попуњавамо колону по колону одозго наниже, можемо добити меморијску сложеност $O(n)$ – временска сложеност остаје $O(n^2)$. Наиме, сваки елемент зависи од елемента у истој врсти у претходној колони и елемента у истој колони у некој од претходних врста, тако да ако колоне попуњавамо одозго наниже, можемо чувати само две узастопне колоне. Заправо, можемо чувати и само једну колону, ако њено попуњавање организујемо тако да се током ажурирања сви елементи пре текуће врсте односе на вредности текуће колоне, а од текуће врсте до краја односе на вредности претходне колоне. Приметимо да се у делу где је $n < s_{max}$, вредности

између две суседне колоне не мењају. Тиме добијамо наредну оптимизовану имплементацију.

```
#include <iostream>
#include <vector>

using namespace std;

int brojParticija(int N) {
  vector<int> dp(N+1, 0);
  dp[0] = 1;
  for (int smax = 1; smax <= N; smax++)
    for (int n = smax; n <= N; n++)
      dp[n] += dp[n-smax];
  return dp[N];
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
  return 0;
}
```

Динамичким програмирањем можемо оптимизовати и случај када се броје партиције са неопадајуће сортираним сабирцима. Прикажимо табелу вредности функције за $n = 8$.

```
n\smin    1  2  3  4  5  6  7  8
1         1  0  0  0  0  0  0  0
2         2  1  0  0  0  0  0  0
3         3  1  1  0  0  0  0  0
4         5  2  1  1  0  0  0  0
5         7  2  1  1  1  0  0  0
6        11  4  2  1  1  1  0  0
7        15  4  2  1  1  1  1  0
8        22  7  3  2  1  1  1  1
```

Табелу можемо попуњавати врсту по врсту. Нажалост, меморијску оптитмизацију је овде теже остварити.

```
#include <iostream>

using namespace std;

const int MAX = 100;

int brojParticija(int N) {
  int br[MAX + 1][MAX + 1] = {0};

  // br[i][j] je broj particija broja i pomocu sabiraka >= smin
  for(int n = 1; n <= N; n++) {
    br[n][n] = 1;
    for(int smin = n-1; smin > 0; smin--)
      br[n][smin] = br[n][smin+1] + br[n-smin][smin];
  }
  return br[N][1];
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
  return 0;
```

}

Задатак: Број цик-цак партиција

Напиши програм који одређује број партиција природног броја n (разбијања на сабирке који су позитивни природни бројеви) таквих да сабирци наизменично расту и опадају (или опадају па расту).

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 5000$), а затим и број a_0 ($1 \leq a_0 \leq n$), који представља први сабирак у партицији.

Излаз: На стандардни излаз исписати остатак при дељењу броја партиција са $10^9 + 7$.

Пример 1

Улаз	*Излаз*
5 2	2

Објашњење

Партције су $2 + 1 + 2$ и $2 + 3$.

Пример 2

Улаз

```
7
3
```

Излаз

```
3
```

Објашњење

Партције су $3 + 1 + 2 + 1$, $3 + 1 + 3$ и $3 + 4$.

Пример 3

Улаз

```
1000
500
```

Излаз

```
562222907
```

Решење

Рекурзивна формулација

Бројање партиција се може вршити рекурзивном функцијом, слично ка у задатку Број партиција. Након постављања вредности одређеног сабирка, рекурзивно се партиционише преостали збир (који се добије умањивањем тренутног збира за постављени сабирак). Наравно, овај поступак је потребно модификовати, тако да се броје само цик-цак партиције (оне у којима сабирци наизменично расту и опадају, тј. опадају и расту).

Пошто је први сабирак фиксиран, можемо размотрити све могућности за други сабирак. Пошто су допуштене и партиције које прво расту, па онда опадају и партиције које прво опадају, па онда расту, други сабирак може бити већи, било мањи од првог, међутим, већ након постављања другог сабирка, за сваки наредни сабирак је једнозначно одређено да ли мора да буде мањи или већи од њему претходног сабирка. Зато ће наша рекурзивна функција поред збира добијати и први сабирак и податак о томе да ли наредни сабирак мора да буде мањи или већи од првог. У почетку вршимо два рекурзивна позива, један који израчунава број партиција када је други сабирак већи, а други када је други сабирак мањи од првог, и коначан резултат добијамо сабирањем њихових резултата (изузетак је случај када је први сабирак у старту једнак збиру, када знамо да постоји само једна, тривијална, партиција).

Излаз из рекурзије представља случај када је збир једнак производу и тада постоји тачно једна таква партиција. У супротном анализирамо све могућности за други сабирак и вршимо даље рекурзивне позиве. Ако други сабирак треба да буде већи од првог, тада разматрамо вредности из интервала $[p + 1, z - p]$, где је p

вредност првог сабирка, а z вредност збира. Ако други сабирак треба да буде мањи од првог, тада разматрамо вредности из интервала $[1, \min(p-1, z-p)]$. Сабирамо бројеве партиција за сваку вредност другог сабирка (по задатом модулу) и враћамо збир.

Дакле, ако обележимо са $r_{z,p}$ број партиција збира z, где је први сабирак p, а други већи од њега, а са $o_{z,p}$ број партиција збира z, где је први сабирак p, а други мањи од њега, важе следеће рекурентне везе.

$$\begin{aligned} r_{z,z} &= 1 \\ o_{z,z} &= 1 \\ r_{z,p} &= \sum_{d=p+1}^{z-p} o_{z-p,d}, \quad p < z \\ o_{z,p} &= \sum_{d=1}^{\min(p-1,z-p)} r_{z-p,d}, \quad p < z \end{aligned}$$

Наравно, све ове збирове треба рачунати по датом модулу.

```
#include <iostream>
#include <vector>
#include <array>
using namespace std;

const int MOD = 1e9 + 7;

int brojCikCakParticija(int prvi, int zbir, bool raste) {
  if (zbir == prvi)
    return 1;
  int broj = 0;
  if (raste)
    for (int sledeci = prvi+1; sledeci <= zbir - prvi; sledeci++)
      broj = (broj + brojCikCakParticija(sledeci, zbir - prvi, false)) % MOD;
  else
    for (int sledeci = 1; sledeci < prvi && sledeci <= zbir - prvi; sledeci++)
      broj = (broj + brojCikCakParticija(sledeci, zbir - prvi, true)) % MOD;
  return broj;
}

int brojCikCakParticija(int prvi, int zbir) {
  if (zbir == prvi)
    return 1;

  return (brojCikCakParticija(prvi, zbir, true) +
          brojCikCakParticija(prvi, zbir, false)) % MOD;
}

int main() {
  int prvi, zbir;
  cin >> zbir >> prvi;
  cout << brojCikCakParticija(prvi, zbir) << endl;
}
```

Могуће је дефинисати и две узајамно рекурзивне функције.

```
#include <iostream>
#include <vector>
#include <array>
using namespace std;
```

```
const int MOD = 1e9 + 7;

int brojCikCakParticijaRaste(int prvi, int zbir);
int brojCikCakParticijaOpada(int prvi, int zbir);

int brojCikCakParticijaRaste(int prvi, int zbir) {
  if (zbir == prvi)
    return 1;
  int broj = 0;
  for (int sledeci = prvi+1; sledeci <= zbir - prvi; sledeci++)
    broj = (broj + brojCikCakParticijaOpada(sledeci, zbir - prvi)) % MOD;
  return broj;
}

int brojCikCakParticijaOpada(int prvi, int zbir) {
  if (zbir == prvi)
    return 1;
  int broj = 0;
  for (int sledeci = 1; sledeci < prvi && sledeci <= zbir - prvi; sledeci++)
    broj = (broj + brojCikCakParticijaRaste(sledeci, zbir - prvi)) % MOD;
  return broj;
}


int brojCikCakParticija(int prvi, int zbir) {
  if (zbir == prvi)
    return 1;

  return (brojCikCakParticijaRaste(prvi, zbir) +
          brojCikCakParticijaOpada(prvi, zbir)) % MOD;
}

int main() {
  int prvi, zbir;
  cin >> zbir >> prvi;
  cout << brojCikCakParticija(prvi, zbir) << endl;
}
```

Мемоизација

Пошто се у рекурзивној формулацији често врше идентични рекурзивни позиви, ефикасност се може поправити динамичким програмирањем. На пример, могуће је применити мемоизацију. За мемоизацију можемо употребити две матрице (једну за вредности $r_{z,p}$, а другу за вредности $o_{z,p}$, а можемо и те две матрице спаковати у тродимензионални низ (чија је последња димензија једнака 2, тако да нпр. индекс 0 одређује вредности $r_{z,p}$, а индекс 1 одређује вредности $o_{z,p}$).

Сложеност се грубо може проценити на следећи начин. Вредности у матрици се рачунају у линеарној сложености, па пошто је димензија матрице квадратна (у односу на полазни збир z), сложеност је $O(z^3)$.

```
#include <iostream>
#include <vector>
#include <array>
using namespace std;

const int MOD = 1e9 + 7;

int brojCikCakParticija(int prvi, int zbir, bool raste,
                        vector<vector<array<int, 2>>>& memo) {
  if (memo[prvi][zbir][raste] != -1)
    return memo[prvi][zbir][raste];
```

```
  if (zbir == prvi)
    return memo[prvi][zbir][raste] = 1;
  int broj = 0;
  if (raste)
    for (int sledeci = prvi+1; sledeci <= zbir - prvi; sledeci++)
      broj = (broj + brojCikCakParticija(sledeci, zbir - prvi, false, memo)) % MOD;
  else
    for (int sledeci = 1; sledeci < prvi && sledeci <= zbir - prvi; sledeci++)
      broj = (broj + brojCikCakParticija(sledeci, zbir - prvi, true, memo)) % MOD;
  return memo[prvi][zbir][raste] = broj;
}

int brojCikCakParticija(int prvi, int zbir) {
  if (prvi == zbir)
    return 1;

  vector<vector<array<int, 2>>> memo(zbir + 1, vector<array<int, 2>>(zbir + 1, {-1, -1}));
  return (brojCikCakParticija(prvi, zbir, true, memo) +
          brojCikCakParticija(prvi, zbir, false, memo)) % MOD;
}

int main() {
  int prvi, zbir;
  cin >> zbir >> prvi;
  cout << brojCikCakParticija(prvi, zbir) << endl;
}
```

Динамичко програмирање навише

Рекурзивна конструкција се може преточити у индуктивну и задатак се може решити динамичким програмирањем навише. Матрице можемо попуњавати врсту по врсту, али услед узајамне зависности вредности у матрицама, чим попунимо једну врсту у матрици $r_{z,p}$, одмах ту врсту морамо попунити у матрици $o_{z,p}$. Приликом попуњавања вредности у текућој врсти имамо на располагању све потребне вредности. Наиме $r_{z,p}$ зависи од вредности облика $o_{z-p,k}$, што је већ израчунато, јер је $p \geq 1$. Слично је и за $o_{z,p}$.

Сложеност попуњавања матрица је $O(z^3)$, где је z почетна вредност збира.

```
#include <iostream>
#include <vector>
using namespace std;

const int MOD = 1e9 + 7;

int brojCikCakParticija(int prvi, int zbir) {
  if (prvi == zbir)
    return 1;

  vector<vector<int>> raste(zbir+1, vector<int>(zbir+1, 0));
  vector<vector<int>> opada(zbir+1, vector<int>(zbir+1, 0));

  for (int z = 1; z <= zbir; z++) {
    for (int p = 1; p < z; p++) {
      raste[z][p] = 0;
      for (int k = p+1; k <= z-p; k++)
        raste[z][p] = (raste[z][p] + opada[z-p][k]) % MOD;
      opada[z][p] = 0;
      for (int k = 1; k <= p-1 && k <= z-p; k++)
        opada[z][p] = (opada[z][p] + raste[z-p][k]) % MOD;
    }
    raste[z][z] = opada[z][z] = 1;
```

```
  }

  return (raste[zbir][prvi] + opada[zbir][prvi]) % MOD;
}

int main() {
  int prvi, zbir;
  cin >> zbir >> prvi;
  cout << brojCikCakParticija(prvi, zbir) << endl;
}
```

Оптимизација коришћењем префиксних збирова

Приметимо да се вредности $r_{z,p}$ и $o_{z,p}$ израчунавају као збир одређених сегмената (узастопних елемената) претходних врста матрице. У задатку Збирови сегмената видели смо да се збирови елемената сегмената могу израчунати ефикасно ако се израчунају збирови префикса низа. Ту технику је могуће применити и у овом задатку и уместо вредности $r_{z,p}$ можемо чувати збир префикса $R_{z,p} = \sum_{k=1}^{p} r_{z,k}$ и слично уместо вредности $o_{z,p}$ можемо чувати вредности $O_{z,p} = \sum_{k=1}^{p} o_{z,k}$. Тада за свако $p < z$ важи:

$$\begin{aligned} r_{z,p} &= \sum_{d=p+1}^{z-p} o_{z-p,d} = O_{z-p,z-p} - O_{z-p,p} \\ R_{z,p} &= R_{z,p-1} + r_{z,p} \\ o_{z,p} &= \sum_{d=1}^{\min(p-1,z-p)} r_{z-p,d} = R_{z-p,\min(p-1,z-p)} \\ O_{z,p} &= O_{z,p-1} + o_{z,p} \end{aligned}$$

Такође, пошто је $r_{z,z} = o_{z,z} = 1$, важи

$$\begin{aligned} R_{z,z} &= R_{z,z-1} + 1 \\ O_{z,z} &= O_{z,z-1} + 1 \end{aligned}$$

Наравно, све ове збирове треба рачунати по датом модулу.

На крају, у случају да је $p < z$, вредности $r_{z,p}$ и $o_{z,p}$ се израчунавају као $O_{z,p} - O_{z,p-1}$ и $R_{z,p} - R_{z,p-1}$ (у супротном, ако је $z = p$, знамо да је резултат 1 и без икаквог рачунања).

Пошто се сада свака вредност у матрици рачуна у времену $O(1)$, а у матрици постоји $O(z^2)$ елемената, сложеност алгоритма је $O(z^2)$.

```
#include <iostream>
#include <vector>
using namespace std;

const int MOD = 1e9 + 7;

int brojCikCakParticija(int prvi, int zbir) {
  if (prvi == zbir)
    return 1;

  vector<vector<int>> zbirRaste(zbir+1);
  for (int i = 0; i <= zbir; i++)
    zbirRaste[i].resize(zbir+1, 0);
  vector<vector<int>> zbirOpada(zbir+1);
  for (int i = 0; i <= zbir; i++)
    zbirOpada[i].resize(zbir+1, 0);
```

```
  for (int z = 1; z <= zbir; z++) {
    for (int p = 1; p < z; p++) {
      if (p < z-p) {
        int raste_zp = (zbirOpada[z-p][z-p] - zbirOpada[z-p][p] + MOD) % MOD;
        zbirRaste[z][p] = (zbirRaste[z][p-1] + raste_zp) % MOD;
      } else
        zbirRaste[z][p] = zbirRaste[z][p-1];

      int opada_zp = zbirRaste[z-p][min(p-1, z-p)];
      zbirOpada[z][p] = (zbirOpada[z][p-1] + opada_zp) % MOD;
    }
    zbirRaste[z][z] = (zbirRaste[z][z-1] + 1) % MOD;
    zbirOpada[z][z] = (zbirOpada[z][z-1] + 1) % MOD;
  }
  int raste = (zbirRaste[zbir][prvi] - zbirRaste[zbir][prvi-1] + MOD) % MOD;
  int opada = (zbirOpada[zbir][prvi] - zbirOpada[zbir][prvi-1] + MOD) % MOD;
  return (raste + opada) % MOD;
}

int main() {
  int prvi, zbir;
  cin >> zbir >> prvi;
  cout << brojCikCakParticija(prvi, zbir) << endl;
}
```

Задатак: Разлагање на збир квадрата

Написати програм којим се одређује минималан број квадрата природних бројева чији је збир једнак датом природном броју n.

Улаз: Прва и једина линија стандардног улаза садржи природан број n ($n \leq 10^5$).

Излаз: На стандардном излазу приказати минималан број квадрата чији је збир једнак броју n.

Пример

Улаз	*Излаз*
72	2

Објашњење

$$72 = 36 + 36$$

Решење

Чувена Лагранжова теорема о збиру четири квадрата нам гарантује да се сваки природан број може представити као збир највише 4 квадрата природних бројева. Решење овог задатка то експериментално доказује за бројеве које будемо тестирали, а видећемо и да познавање Лагранжове теореме омогућава ефикасније решење.

Овај задатак има одређених сличности са одређивањем партиција броја које смо разматрали у задацима Све партиције и Број партиција. Размотримо рекурзивно решење грубом силом. Размотримо све могућности за први сабирак у разлагању на збир квадрата броја n. Најмањи кандидат за први сабирак је 1, а највећи је квадрат броја $\lfloor \sqrt{n} \rfloor$. За сваки могући сабирак s^2 рекурзивно одређујемо разлагање броја $n - s^2$ на збир квадрата и тражимо минимум добијених вредности увећаних за 1. Ако не узмемо у обзир Лагранжову теорему, минимум можемо иницијализовати на n, јер се сваки број може разложити на збир n јединица које заиста представљају квадрате природних бројева. Излаз из рекурзије представља случај $n = 0$ и тада се враћа 0 (није потребно додати ни један сабирак да би се направила 0). Дакле, број квадрата $b(n)$ се може изразити следећим рекурентним везама.

$$b(0) = 0$$
$$b(n) = \min_{1 \leq s^2 \leq n} (1 + b(n - s^2))$$

```
#include <iostream>
#include <algorithm>

using namespace std;

int brojKvadrata(int n) {
  if (n == 0)
    return 0;
  int br = n;
  for (int s = 1; s*s <= n; s++)
    br = min(br, brojKvadrata(n-s*s) + 1);
  return br;
}

int main() {
  int n;
  cin >> n;
  cout << brojKvadrata(n) << endl;
  return 0;
}
```

Претходни једноставан алгоритам не узима у обзир комутативност сабирања тако да се разлагање на исте сабирке разматра више пута (на пример, 10 се разлаже и као $1^2 + 3^2$ и као $3^2 + 1^2$). Процедура може да буде оптимизована ако се наметне додатни услов да је сваки наредни сабирак већи или једнак од претходног. Зато функцију можемо проширити параметром који представља доњу границу допуштених сабирака ($b(n, m)$ представља најмањи број квадрата којима се може изразити број n, када је сваки квадрат већи или једнак од m^2). Када у збир укључимо сабирак s^2, тада се у рекурзивном позиву граница сабирка подиже на s^2.

$$b(0, m) = 0$$
$$b(n, m) = \min_{m^2 \leq s^2 \leq n} (1 + b(n - s^2, s))$$

Овим се рекурзивна претрага осетно убрзава, али и даље остаје спора, јер се исти рекурзивни позиви понављају више пута.

```
#include <iostream>
#include <algorithm>

using namespace std;

int brojKvadrata(int n, int max) {
  if (n == 0)
    return 0;
  int br = n;
  for (int s = max; s*s <= n; s++)
    br = min(br, brojKvadrata(n-s*s, s) + 1);
  return br;
}

int main() {
  int n;
  cin >> n;
  cout << brojKvadrata(n, 1) << endl;
```

```
  return 0;
}
```

Боље решење добијамо када применимо динамичко програмирање. С обзиром на то да би у случају меморисања два параметра меморијски захтеви били недопустиво велики, одустајемо од варијанте у којој се захтева уређеност сабирака и враћамо се на основну варијанту у којој се не обраћа пажња на комутативност. У том случају рекурзивна функција има један параметар и мемоизацију можемо вршити само помоћу једног низа. Задатак можемо једноставно решити динамичким програмирањем навише. На место 0 у низу уписујемо 0, а затим низ попуњавамо редом, одоздо навише, све док се не израчуна вредност за тражени број n.

За одређивање броја делилаца броја N, спољашња петља се извршава N пута, а унутрашња $\sqrt{n}$ пута, где је n текућа вредност бројача спољашње петље, тако да је сложеност $O(\sqrt{1}+\sqrt{2}+\ldots+\sqrt{N})$. Лако се покаже да је ово одозго ограничено и да важи да је $O(N^{\frac{3}{2}})$, а може се показати и да је ова асимпотска граница егзактна тј. да је сложеност $\Theta(N^{\frac{3}{2}})$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int brojKvadrata(int N) {
  vector<int> br(N+1);
  br[0] = 0;
  for (int n = 1; n <= N; n++) {
    br[n] = n;
    for(int s = 1; s*s <= n; s++)
        br[n] = min(br[n], br[n-s*s] + 1);
  }
  return br[N];
}

int main() {
  int n;
  cin >> n;
  cout << brojKvadrata(n) << endl;
  return 0;
}
```

Решење је могуће директно засновати на примени Лагранжеве теореме, која нам гарантује да је тражени број увек највише 4. Исцрпном претрагом проверавамо да ли је број могуће разложити на највише 3 квадрата и ако то не успемо враћамо резултат 4. Проверу вршимо помоћу две угнежђене петље у којима набрајамо парове вредности (i,j) такве да је $i \leq j$ и $i^2+j^2 \leq n^2$. За сваки пар проверавамо да ли је $n^2-i^2-j^2$ потпун квадрат. Ако то важи за $i=j=0$, сам број је потпун квадрат, ако важи за $i=0$ и $j \neq 0$ тада се број разбија на збир два квадрата, а ако важи за $i \neq 0$ и $j \neq 0$, тада се број разбија на три квадрата. Ако ни један пар не задовољава претходни услов, онда се број не може разбити на збир највише три квадрата и враћамо резултат 4 (на основу Лагранжове теореме).

Петља по i тече од 0 до $\sqrt{n}$, а по j тече од i до $\sqrt{n-i^2}$, док у телу унутрашње петље проверу да ли је остатак потпун квадрат вршимо у константној сложености, тако да је сложеност целог алгоритма овај пут одозго јасно ограничена са n тј. износи $O(n)$. И емпиријски се показује да је овај приступ ефикаснији од приступа заснованог на динамичком програмирању. Ипак, предност динамичког програмирња је то што је то решење веома опште, док се у овом решењу користи веома специфично доменско знање (Лагранжова теорема).

```
#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;

int brojKvadrata(int n) {
  for (int i = 0; i*i <= n; i++)
```

```
    for (int j = i; j*j <= n-i*i; j++) {
      int k = n - i*i - j*j;
      int kk = (int)sqrt(k);
      if (kk * kk == k) {
        if (j == 0)
          return 1;
        else if (i == 0)
          return 2;
        else
          return 3;
      }
    }
  return 4;
}

int main() {
  int n;
  cin >> n;
  cout << brojKvadrata(n) << endl;
  return 0;
}
```

Задатак: Број начина разлагања на збир различитих n-тих степена

Написати програм којим се за дате природне бројеве x и n приказује на колико начина број x може да се изрази као збир n-тих степена различитих природних бројева.

Улаз: У првој линији стандардног улаза налази се природан број x ($x \leq 10000$). Друга линија стандардног улаза садржи природан број n ($n < 10$).

Излаз: На стандардном излазу приказати у једној линији тражени број начина.

Пример 1

Улаз *Излаз*

```
10        1
2
```

Објашњење

$$10 = 1^2 + 3^2$$

Пример 2

Улаз

```
1729
3
```

Излаз

```
4
```

Објашњење

$$1729 = 1^3 + 12^3 = 1^3 + 3^3 + 4^3 + 5^3 + 8^3 + 10^3 = 1^3 + 6^3 + 8^3 + 10^3 = 9^3 + 10^3$$

Решење

Задатак можемо решити рекурзивно, слично као задатак Број партиција. Функција прима број x који треба разложити као и најмању основу o чији се степен може јавити као сабирак у том разлагању, да би се обезбедило да се бројеви не понављају. У почетку је та најмања основа 1, а током рекурзије она ће се повећавати, да би се забранило укључивање мањих бројева који већ обрађени. Један излаз из рекурзије је када се разлаже број $x = 0$ - он се увек разлаже на јединствен начин (не узевши ни један сабирак). Други излаз из рекурзије

је када је n-ти степен најмање основе већи од броја који се разлаже ($o^n > x$) и тада знамо да не постоји начин да се број разложи. У супротном, постоје две могућности: или се n-ти степен основе o^n јавља као сабирак у разлагању броја x или се не јавља. Рекурзивно израчунавамо број начина да се $x - o^n$ разложи, при чему је тада најмања основа број $o + 1$ и тај број сабирамо са бројем начина да се број x разложи ако је најмања основа $o + 1$.

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

int stepen(int a, int n) {
  if (n == 0)
    return 1;
  else if (n % 2 == 0)
    return stepen(a * a, n / 2);
  else
    return a * stepen(a, n - 1);
}

long long brNacina(int x, int n, int minOsnova) {
  if (x == 0)
    return 1;

  int st = stepen(minOsnova, n);
  if (st > x)
    return 0;

  return brNacina(x - st, n, minOsnova + 1) + brNacina(x, n, minOsnova + 1);
}

long long brNacina(int x, int n) {
  return brNacina(x, n, 1);
}

int main() {
   int x, n;
   cin >> x >> n;
   cout << brNacina(x, n) << endl;
   return 0;
}
```

Пошто у рекурзивном решењу долази до преклапања рекурзивних позива, потребно је употребити неки облик динамичког програмирања да би се решење временски оптимизовало (по цену употребе веће количине меморије). Ако употребимо мемоизацију, у матрици ћемо памтити резултате раније обрађених рекурзивних позива. Анализом рекурзивне функције можемо установити да параметар x може имати све вредности између 0 и x (полазног броја који се разлаже), док параметар o (најмања основа) може имати све вредности између 1 и $\lceil \sqrt{x} \rceil$, па на основу тога знамо и које су димензије матрице потребне.

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

int stepen(int a, int n) {
  if (n == 0)
    return 1;
  else if (n % 2 == 0)
```

```
    return stepen(a * a, n / 2);
  else
    return a * stepen(a, n - 1);
}

long long brNacina(int x, int n, int minOsnova, vector<vector<long long>>& memo) {
  if (memo[x][minOsnova] != -1)
    return memo[x][minOsnova];

  if (x == 0)
    return memo[x][minOsnova] = 1;

  int st = stepen(minOsnova, n);
  if (st > x)
    return memo[x][minOsnova] = 0;

  return memo[x][minOsnova] =
    brNacina(x - st, n, minOsnova + 1, memo) +
    brNacina(x, n, minOsnova + 1, memo);
}

long long brNacina(int x, int n) {
  int ntiKoren = (int)ceil(pow(x, 1.0/n));
  vector<vector<long long>> memo(x + 1, vector<long long>(ntiKoren+1, -1));
  return brNacina(x, n, 1, memo);
}

int main() {
   int x, n;
   cin >> x >> n;
   cout << brNacina(x, n) << endl;
   return 0;
}
```

Задатак је могуће решити и попуњавањем целе матрице врсту по врсту, техником динамичког програмирања навише. У прву врсту (која одговара случају $x = 0$) уписаћемо све јединице (јер се нула разлаже на јединствен начин). Затим ћемо попуњавати наредне врсте и то здесна налево (јер за исто x вредност за неку основу o зависи од вредности за основу $o+1$). Док је степен основе строго већи од текућег броја, у врсту ћемо уписиваћемо нуле (алтернативно, целу матрицу можемо иницијализовати на нулу, па ову петљу прескочити). Када се основа довољно смањи, тада ћемо у врсту уписивати збир наредног елемента врсте и наредног елемента из неке претходне врсте која одговара броју добијеном одузимању степена основе од текућег броја.

Приметимо да елементи у текућој врсти не зависе само од елемената из претходне врсте, па не можемо извршити меморијску оптимизацију тако што бисмо уместо матрице чували само једну или две узастопне врсте.

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

int stepen(int a, int n) {
  if (n == 0)
    return 1;
  else if (n % 2 == 0)
    return stepen(a * a, n / 2);
  else
    return a * stepen(a, n - 1);
}
```

```
long long brNacina(int X, int n) {
  int ntiKoren = (int)ceil(pow(X, 1.0/n));
  vector<vector<long long>> dp(X + 1, vector<long long>(ntiKoren + 1, 0));
  for (int osnova = 1; osnova <= ntiKoren; osnova++)
    dp[0][osnova] = 1;
  for (int x = 1; x <= X; x++) {
    int osnova = ntiKoren + 1;
    int st;
    do {
      osnova--;
      st = stepen(osnova, n);
    } while (st > x);
    while (osnova >= 0) {
      dp[x][osnova] = dp[x-st][osnova+1] + dp[x][osnova+1];
      osnova--;
      st = stepen(osnova, n);
    }
  }
  return dp[X][1];
}

int main() {
   int x, n;
   cin >> x >> n;
   cout << brNacina(x, n) << endl;
   return 0;
}
```

Задатак: Дигитални бројач

$2n$-то цифрени дигитални бројач који одбројава од $000\ldots000$ до $999\ldots999$ емитује звучни сигнал сваки пут када је сума првих n цифара једнака суми последњих n цифара. На пример, за шестоцифрени дигитални бројач звучни сигнал се пушта за $000000, 001001, 001010, \ldots, 999999$. Написати програм који ће одредити колико пута ће бити емитован звучни сигнал.

Улаз: У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 5$).

Излаз: На стандарном излазу приказати колико постоји $2n$-цифрених бројева са траженим својством.

Пример

Улаз	*Излаз*
3	55252

Решење

Наивна решења

Директно решење је да се у циклусу прођу сви бројеви од $00\ldots0$ до $99\ldots9$, да се за сваки број одреди збир леве и десне половине, и ако су они једнаки, да се увећа бројач (у овом решењу се комбинују алгоритам бројања филтриране серије као у задатку Просек одличних и алгоритам сабирања цифара броја као у задатку Број и збир цифара броја). Сложеност овог алгоритма је $O(2n \cdot 10^{2n})$.

```
#include <iostream>
#include <cmath>

using namespace std;

// odredjuje i uklanja poslednjih n cifara broja x
int zbirNPoslednjihCifara(long long& x, int n) {
  int zbir = 0;
  for (int i = 0; i < n; i++) {
    zbir += x % 10;
```

```
    x /= 10;
  }
  return zbir;
}

// odredjuje zbirove prve i druge polovine broja x koji ima 2n cifara
void zbiroviCifara(long long x, int n,
                   int& zbirPrvePolovine, int& zbirDrugePolovine) {
  zbirDrugePolovine = zbirNPoslednjihCifara(x, n);
  zbirPrvePolovine = zbirNPoslednjihCifara(x, n);
}

int main() {
  int n;
  cin >> n;
  long long ukupnoBrojeva = 0;
  long long max = (long long)pow(10, 2*n) - 1;
  for (long long i = 0; i <= max; i++) {
    int zbirPrvePolovine, zbirDrugePolovine;
    zbiroviCifara(i, n, zbirPrvePolovine, zbirDrugePolovine);
    if (zbirPrvePolovine == zbirDrugePolovine)
      ukupnoBrojeva++;
  }
  cout << ukupnoBrojeva << endl;

  return 0;
}
```

Мало једноставније и ефикасније решење је да се лева и десна половина броја набрајају у две угнежђене петље и броји колико пута ће збир цифара спољашње циклусне променљиве бити једнак збиру цифара унутрашње циклусне променљиве. Ако се збир леве половине броја рачуна само једном по извршавању целокупне унутрашње петље добијамо мало побољшање у односу на претходни алгоритам, али је сложеност и даље $O(n \cdot 10^{2n})$.

```
#include <iostream>
#include <cmath>

using namespace std;

int zbirCifara(long long x) {
  int zbir = 0;
  while (x > 0) {
    zbir += x % 10;
    x /= 10;
  }
  return zbir;
}

int main() {
  int n;
  cin >> n;
  long long ukupnoBrojeva = 0;
  long long max = (int)pow(10, n) - 1;
  for (int prvaPolovina = 0; prvaPolovina <= max; prvaPolovina++) {
    int zbirPrvePolovine = zbirCifara(prvaPolovina);
    for (int drugaPolovina = 0; drugaPolovina <= max; drugaPolovina++) {
      int zbirDrugePolovine = zbirCifara(drugaPolovina);
      if (zbirPrvePolovine == zbirDrugePolovine)
        ukupnoBrojeva++;
```

```
    }
  }

  cout << ukupnoBrojeva << endl;

  return 0;
}
```

Број n-тоцифрених бројева датог збира

Пошто лева и десна половина треба да имају исту суму (на пример, s), и једна и друга половина ће бити n-тоцифрени бројеви чија је сума s. Сума цифара n-тоцифреног броја узима вредности од 0 (за број $00\ldots0$) до $n \cdot 9$ (за број $99\ldots9$). За свако такво s треба да одредимо на колико разних начина можемо да одаберемо два броја чија је сума цифара s. Означимо са b_s број n-тоцифрених бројева којима је сума цифара s (број између 0 и $9 \cdot n$). Дакле и прву и другу половину можемо изабрати на b_s начина (пошто оне не морају бити различити), тако да постоји укупно b_s^2 бројева чија лева и десна половина имају збир s. Зато ће тражени број бити једнак суми $b_0^2 + b_1^2 + ... + b_{9n}^2$.

Остаје питање како одредити број n-тоцифрених бројева чији је збир s.

Бројање бројева датог збира

Један начин је да се уведе низ бројача (асоцијативни низ, слично као у задатку Фреквенција знака) - по један за сваку вредност c да се у петљи обиђу сви бројеви од 0 до $10^n - 1$, да се за сваки одреди збир цифара и да се увећа бројач бројева добијеног збира. Сложеност овог алгоритма је $n \cdot 10^n$, што је и укупна сложеност алгоритма (јер је рачунање коначне суме само $O(n)$).

```
#include <iostream>
#include <vector>

using namespace std;

// odredjuje zbir cifara broja n
int SumaCifara(long long n) {
  int s = 0;
  while (n > 0) {
    s += n % 10;
    n /= 10;
  }
  return s;
}

// odredjuje stepen x^n
long long Stepen(int x, int n) {
  long long s = 1;
  for (int i = 0; i < n; i++)
    s *= x;
  return s;
}

int main() {
  int n;
  cin >> n;

  // broj brojeva sa datom sumom cifara
  vector<long long> brojBrojevaDateSumeCifara(9*n + 1, 0);

  // za sve n-tocifrene brojeve izmedju 00..0 i 99...9
  long long st10 = Stepen(10, n);
  for (long long broj = 0; broj < st10; broj++)
    // odredjujemo sumu cifara i uvecavamo broj brojeva
```

```
    // sa tom sumom cifara
    brojBrojevaDateSumeCifara[SumaCifara(broj)]++;

  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int suma = 0; suma <= 9*n; suma++) {
    // postoji b n-tocifrenih brojeva koji imaju sumu cifara suma
    long long b = brojBrojevaDateSumeCifara[suma];
    // levu polovinu broja mozemo odabrati na b nacina i desnu
    // polovinu na b nacina, tako da ukupno takvih brojeva ima b*b
    ukupnoBrojeva += b * b;
  }

  cout << ukupnoBrojeva << endl;
  return 0;
}
```

Бројање партиција динамичким програмирањем

Други начин да се одреде вредности n-тоцифрених бројева чија је сума b_s је да се примени рекурзивно решење по броју цифара, слично какво смо видели у задацима Сви n-тоцифрени бројеви са датим збиром цифара и Број партиција.

Уведимо стога ознаку $b_{k,s}$ за број k-тоцифрених бројева чија је сума цифара s. Ако је $k = 1$, бројева $b_{1,s}$ има 1 за $0 \leq s \leq 9$, тј. 0 у супротном. Ако је $k > 1$ тада на прво место можемо поставити било коју цифру c од 0 до 9 (тј. до s ако је $s < 9$). Када њу поставимо, остале цифре можемо поставити на $b_{k-1,s-c}$ начина. Дакле,

$$b_{k,s} = \sum_{c=0}^{\min(s,9)} b_{k-1,s-c}.$$

Пошто се у овој рекрузивној формулацији рекурзивни позиви преклапају, потребно је употребити технику динамичког програмирања да би се решење убрзало. Једна могућност је да употребимо мемоизацију.

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

// broj nacina da se napravi broj koji ima brojCifara cifara i kome je
// zbir cifara jednak zbirCifara
long long brojParticija(int zbirCifara, int brojCifara) {
  // koristimo memoizaciju da bismo izbegli da vise puta racunamo
  // jedan te isti rezultat - najjednostavnija (ali ne i
  // najefikasnija) implementacija cuva ranije rezultate u recniku
  static map<pair<int, int>, long long> memo;
  // proveravamo da li smo vec racunali ovaj rezultat i ako jesmo
  // vracamo raniji rezultat
  auto p = make_pair(zbirCifara, brojCifara);
  if (memo.find(p) != memo.end())
    return memo[p];

  if (brojCifara == 1)
    // brojevi izmedju 0 i 9 se mogu na jedan nacin predstaviti kao
    // zbir jedne cifre, a veci brojevi od toga se ne mogu predstaviti
    return memo[p] = zbirCifara < 10 ? 1 : 0;
  else {
```

```
    long long rezultat = 0;
    // na prvo mesto postavljamo redom jednu po jednu cifru
    for (int cifra = 0; cifra <= 9 && cifra <= zbirCifara; cifra++)
      // rekurzivno racunamo broj particija preostalog zbira sa jednom
      // cifrom manje
      rezultat += brojParticija(zbirCifara - cifra, brojCifara - 1);
    return memo[p] = rezultat;
  }
}

int main() {
  int n;
  cin >> n;

  long long ukupnoBrojeva = 0;
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++) {
    long long b = brojParticija(zbirCifara, n);
    ukupnoBrojeva += b * b;
  }

  cout << ukupnoBrojeva << endl;
  return 0;
}
```

Уместо мемоизације можемо употребити и динамичко програмирање навише. Вредности $b_{k,s}$ можемо чувати у матрици, чије попуњавање почињемо од првог реда (за $k = 1$) и попуњавамо је затим ред по ред, за растуће вредности k.

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

// alociramo matricu dimenzije m x n
vector<vector<long long>> alociraj(int m, int n) {
  vector<vector<long long>> rezultat(m);
  for (int i = 0; i < m; i++)
    rezultat[i].resize(n);
  return rezultat;
}

// vraca matricu koja za svaki brojCifara od 1 do maxBrojCifara i
// svaki zbirCifara od 1 do maxZbirCifara na mestu
// [brojCifara][zbirCifara] sadrzi broj brojeva koji imaju brojCifara
// cifara i zbir cifara jednak broju zbirCifara
vector<vector<long long>> brojParticija(int maxBrojCifara, int maxZbirCifara) {
  vector<vector<long long>> dp = alociraj(maxBrojCifara + 1, maxZbirCifara + 1);

  // brojevi izmedju 0 i 9 se mogu na jedan nacin predstaviti kao zbir
  // jedne cifre, a veci brojevi od toga se ne mogu predstaviti
  for (int zbirCifara = 0; zbirCifara <= maxZbirCifara; zbirCifara++)
    dp[1][zbirCifara] = zbirCifara < 10 ? 1 : 0;

  // odredjujemo broj razlaganja na vise cifara
  for (int brojCifara = 2; brojCifara <= maxBrojCifara; brojCifara++) {
    // zbir moramo da obilazimo unazad da bismo mogli da koristimo
    // samo jedan niz
    for (int zbirCifara = 0; zbirCifara <= maxZbirCifara; zbirCifara++) {
```

```
      dp[brojCifara][zbirCifara] = 0;
      // na prvo mesto postavljamo redom jednu po jednu cifru
      for (int cifra = 0; cifra <= 9 && cifra <= zbirCifara; cifra++)
        // rekurzivno racunamo broj particija preostalog zbira sa jednom
        // cifrom manje
        dp[brojCifara][zbirCifara] += dp[brojCifara - 1][zbirCifara - cifra];
    }
  }
  return dp;
}

int main() {
  int n;
  cin >> n;

  // za svaki broj od 0 do 9*n nas zanima koliko postoji razlicitih
  // n-tocifrenih brojeva ciji je zbir cifara taj broj
  vector<vector<long long>> dp = brojParticija(n, 9*n);

  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++)
    // postoji dp[n][zbirCifara] nacina da napravimo levu polovinu i
    // dp[n][zbirCifara] nacina da napravimo desnu polovinu broja
    // tj. dp[zbirCifara]*dp[zbirCifara] nacina da odaberemo broj kome
    // i leva i desna polovina imaju zbir cifara zbirCifara
    ukupnoBrojeva += dp[n][zbirCifara] * dp[n][zbirCifara];

  cout << ukupnoBrojeva << endl;
  return 0;
}
```

Приметимо да се за израчунавање вредности у реду k користе само вредности у реду $k-1$. Зато није потребно чувати целу матрицу, већ само њену претходну врсту. Додатно, ако приметимо да за израчунавање вредности $b_{k,s}$ нису потребне вредности из претходног реда за збирове $s' > s$, довољно је да податке чувамо само у једном низу у којем ћемо вредности наредног реда од вредности претходног добијати тако што ћемо кренути од краја и ажурирати једну по једну вредност. Пошто $b_{k,s}$ зависи од $b_{k-1,s}$, потребна нам је једна помоћна променљива у којој ћемо израчунати резултат пре него што га упишемо на место s у низу или вредност можемо добити тако што вредност на месту s која одговара цифри 0 увећамо за вредности на позицијама $s-c$ које одговарају цифрама 1, 2 и тако даље.

Димензија матрице је $n \times 9n$ тј. $O(n^2)$, што је и време потребно за њено попуњавање. Укупна сложеност овог приступа је такође $O(n^2)$ јер је након попуњавања матрице потребно још само $O(n)$ операција. Пошто није потребно чувати целу матрицу, већ само један текући ред, додатна меморија је $O(n)$.

```
#include <iostream>
#include <vector>

using namespace std;

// za svaki broj iz [0, maxZbir] odredjujemo broj nacina da se taj
// broj predstavi kao zbir brojCifara cifara (pri cemu se u obzir
// uzima i redosled cifara)
vector<long long> brojParticija(int brojCifara, int maxZbir) {
  // resenje gradimo dinamickim programiranje
  vector<long long> dp(maxZbir + 1);

  // brojevi izmedju 0 i 9 se mogu na jedan nacin predstaviti kao zbir
  // jedne cifre, a veci brojevi od toga se ne mogu predstaviti
```

```
  for (int zbir = 0; zbir <= maxZbir; zbir++)
    dp[zbir] = zbir < 10 ? 1 : 0;

  // odredjujemo broj razlaganja na vise cifara
  for (int cifara = 2; cifara <= brojCifara; cifara++) {
    // zbir moramo da obilazimo unazad da bismo mogli da koristimo
    // samo jedan niz
    for (int zbirCifara = maxZbir; zbirCifara >= 0; zbirCifara--) {
      // na prvo mesto postavljamo redom jednu po jednu cifru
      for (int cifra = 1; cifra <= 9 && cifra <= zbirCifara; cifra++)
        // rekurzivno racunamo broj particija preostalog zbira sa jednom
        // cifrom manje
        dp[zbirCifara] += dp[zbirCifara - cifra];
    }
  }
  return dp;
}

int main() {
  int n;
  cin >> n;

  // za svaki broj od 0 do 9*n nas zanima koliko postoji razlicitih
  // n-tocifrenih brojeva ciji je zbir cifara taj broj
  vector<long long> bp = brojParticija(n, 9*n);

  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++) {
    // postoji bp[zbirCifara] nacina da napravimo levu polovinu i
    // bp[zbirCifara] nacina da napravimo desnu polovinu broja
    // tj. bp[zbirCifara]*bp[zbirCifara] nacina da odaberemo broj kome
    // i leva i desna polovina imaju zbir cifara zbirCifara
    ukupnoBrojeva += bp[zbirCifara] * bp[zbirCifara];
  }
  cout << ukupnoBrojeva << endl;
  return 0;
}
```

Задатак: Низови 0 и 1 без две суседне 1 - редни број

Посматрајмо низове дужине n које садрже само елементе 0 или 1, а који не садрже два суседна елемента 1. Они се могу поређати лексикографски. На пример сви такви низови дужине 3 поређани у лексикографском поретку су 000, 001, 010, 100, 101. Напиши програм који може да израчуна редни број датог низа у лексикографском поретку, и који за дати број k може да одреди низ који је k-ти по реду у лексикографском поретку (бројање креће од нуле).

Улаз: Прва линија стандардног улаза садржи низ s који садржи само нуле и јединице, али не садржи узастопне јединице (елементи су исписани један до другог, без размака). Дужина низа је између 2 и 35. Наредна линија садржи број n ($2 \leq n \leq 35$) као и природан број k, раздвојене размаком.

Излаз: На стандардном излазу у првој линији редни број низа s, а у другој линији елементе низа дужине n чији је редни број k (или текст `ne postoji` ако је број k већи од највећег редног броја низова дужине n).

Пример 1

Улаз	*Излаз*
01001 5 35	6 ne postoji

Пример 2

Улаз	*Излаз*
01001010 8 35	28 10000001

Решење

Број низова без суседних јединица дате дужине

Анализирајмо прво колико постоји низова који немају две узастопне једнице и који су дужине n. Само је празан низ дужине 0. Низови дужине 1 су 0 и 1. Низови дужине 2 су 00, 01 и 10. Низови дужине 3 су 000, 001, 010, 100 и 101. Приметимо да смо низове дужине 3 могли да добијемо тако да на све низове дужине 2 упишемо нулу на почетак и тако што на низове дужине 1 упишемо 10 на почетак. Ово правило се наставља. Све низове дужине 4 добијамо тако што додамо 0 на почетак свих низова дужине 3 (то су елементи 0000, 0001, 0010, 0010 и 0101) или тако што додамо 10 на почетак свих низова дужине 2 (то су низови 1000, 1001, 1010). Овакав начин изградње низова нам указује на то да је број низова дужине n једнак збиру броја низова дужине $n-1$ и броја низова дужине $n-2$, тј. представља Фибоначијев низ.

Нагласимо и да се функција за израчунавање броја низова мора реализовати уз помоћ мемоизације или динамичког програмирања навише да би се избегло понављање истих рекурзивних позива и избегла експоненцијална сложеност.

Редни број датог низа

Покушајмо сада да на основу датог низа одредимо његов редни број у лексикографском редоследу. Претпоставимо да имамо низ дужине n којем су позиције нумерисане слева надесно. Ако се на позицији i налази јединица, то значи да су сви низови који на позицијама мањим од i имају цифре исте као наш низ, а на позицији i имају нулу лексикографски испред нашег низа. Пошто се заправо варирају све цифре иза позиције i, а њих има $n-i-1$, редни број треба увећати за укупан број низова без узастопних јединица дужине $n-i-1$. Сабирањем ових бројева за све позиције i на којима се налази јединица у полазном низу добијамо његов редни број у лексикографском редоследу.

На пример, ако имамо низ 10101, дужина је $n=5$, а јединице се налазе на позицијама 0, 2 и 4, па се редни број се добија сабирањем броја низова дужине 4, броја низова дужине 2 и броја низова дужине 0, што је на основу претходне анализе једнако $8+3+1=12$. Заиста, ово је највећи низ у лексикографском редоследу и испред њега се налазе низови 00000, 00001, 00010, 00100, 00101, 01000, 01001, 01010, 10000, 10001, 10010, 10100, 10101.

Низ датог редног броја

Одређивање низа на основу редног броја вршимо по сличном поступку. Кључни увид је да у лексикографском поретку свих низова дужине n они чији је редни број строго мањи од броја низова дужине $n-1$, почињу нулом, док остали почињу јединицом. Дакле, почетна цифра у низу дужине n је нула ако је k строго мање од броја низова дужине $n-1$, а један у супротном. Након тога, уклањамо прву цифру и настављамо поступак. Дакле, ако је прва цифра 0, онда тражимо низ са редним бројем k међу низовима дужине $n-1$, а ако је прва цифра 1, онда се од броја k може одузети број низова дужине $n-1$, и потражити низ са тим редним бројем међу низовима дужине $n-1$.

На пример, ако треба да оредимо низ са редним бројем 11 међу низовима дужине 5, прву цифру постављамо на 1 јер постоји 8 низова дужине 4, а затим одређујемо низ са редним бројем 3 међу низовима дужине 4. Пошто постоји 5 низова дужине 3, наредна цифра је 0 и треба да одредимо низ са редним бројем 3 међу низовима дужине 3. Пошто постоји три низа дужине 2, наредна цифра је 1 и треба да одредимо низ са редним бројм 0 међу низовима дужине 2. Постоје 2 низа дужине 1, па наредну цифру постављамо на 0 и након тога одређујемо низ са редним бројем 0 међу низовима дужине 1. Постоји један низ дужине 0, па је наредна цифра 0 и одређујемо низ са редним бројем 1 међу низовима дужине 0. То је празан низ, чиме се поступак завршава и добијамо коначан резултат 10100.

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int brojNizova(int n) {
  const int MAKS_DUZINA = 35;
  static vector<int> _memo(MAKS_DUZINA, 0);

  if (_memo[n] != 0)
    return _memo[n];
```

```
  if (n == 0) return _memo[0] = 1;
  if (n == 1) return _memo[1] = 2;
  return _memo[n] = brojNizova(n-1) + brojNizova(n-2);
}

int redniBrojNiza(string s) {
  int n = s.length();
  int rb = 0;
  for (int i = 0; i < n; i++)
    if (s[i] == '1')
      rb += brojNizova(n - i - 1);
  return rb;
}

bool nizDatogRednogBroja(int n, int rb, string &s) {
  if (rb >= brojNizova(n))
    return false;
  s.resize(n);
  for (int i = 0; i < n; i++)
    if (rb >= brojNizova(n - i - 1)) {
      s[i] = '1';
      rb = rb - brojNizova(n - i - 1);
    } else
      s[i] = '0';
  return true;
}

int main() {
  string s1;
  cin >> s1;
  cout << redniBrojNiza(s1) << endl;

  int n, rb;
  string s2;
  cin >> n >> rb;
  if (nizDatogRednogBroja(n, rb, s2))
    cout << s2 << endl;
  else
    cout << "ne postoji" << endl;

  return 0;
}
```

Задатак: Број начина декодирања

Текст који се састоји само од великих слова енглеске абецеде је кодиран тако што је свако слово замењено његовим редним бројем у абецеди. На пример, текст BABAC је кодиран низом цифара 21213. Међутим, пошто између цифара није прављен размак, декодирање није једнозначно. На пример, 21213 може представити BABAC, али и BAUC, BABM, BLAC, BLM, UBAC, UUC, UBM. Напиши програм који одређује број начина на који је могуће декодирати унети низ цифара.

Улаз: Прва линија стандардног улаза садржи низ цифара добијених кодирањем неког текста - низ има највише 100 цифара.

Излаз: На стандардни излаз исписати број начина да се тај низ декодира (претпоставити да тај број може да стане у 64-битан неозначен цео број).

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
21213	8	1111111111	89	5555555555	1

Пример 4		**Пример 5**	
Улаз	*Излаз*	*Улаз*	*Излаз*
1010101010	1	1111011111	24

Решење

Рекурзивна формулација

Ако кренемо да разматрамо низ цифара од његовог почетка, можемо да издвојимо његову прву цифру, да је декодирамо и затим да декодирамо суфикс добијен његовим уклањањем, или да издвојимо прве две цифре, декодирамо њих и затим да декодирамо суфикс добијен њиховим уклањањем. Први начин је могућ ако низ није празан и ако је прва цифра различита од нуле, док је други могућ ако низ има бар две цифре, ако је прва цифра различита од нуле, а те две цифре граде број између 1 и 26. Ово сугерише рекурзивно решење у којем је параметар рекурзије позиција i, а рекурзивна функција враћа број начина да се декодира суфикс који почиње на позицији i. Излаз из рекурзије је случај празног низа ($i = n$) који се може декодирати на један једини начин (празном ниском).

У овом рекурзивном решењу постојаће веома извесно велики број идентичних рекурзивних позива (тј. позива за исту вредност улазног низа). Нпр. ако низ почиње са 11 тада ће се његов суфикс разматрати и када се посебно декодира свака од јединица и када се декодира пар цифара који гради број 11. Ово доводи до непотребног увећања сложености, а као што знамо, решење долази у облику динамичког програмирања

```
#include <iostream>
#include <string>
#include <limits>

using namespace std;

typedef unsigned long long count_t;

count_t brojNacinaDekodiranja(const string& s) {
  if (s == "")
    return 1;
  count_t res = 0;
  if (s[0] != '0')
    res += brojNacinaDekodiranja(s.substr(1));
  if (s.length() >= 2 && s[0] != '0' && 10*(s[0] - '0') + s[1] - '0' <= 26)
    res += brojNacinaDekodiranja(s.substr(2));
  return res;
}

int main() {
  string s;
  cin >> s;
  cout << brojNacinaDekodiranja(s) << endl;
  return 0;
}
```

Мемоизација

Један од начина да смањимо број рекурзивних позива и непотребна израчунавања је мемоизација. У помоћном низу памтимо до сада израчунате резултате (на позицији i памтимо вредност рекурзивног позива за параметар i). Приликом иницијализације све вредности у низу постављамо на -1 (ако радимо са неозначеним бројевима, уместо -1 употребљава се највећа вредност која се може представити одговарајућим типом). На почетку рекурзивне функције проверавамо да ли је вредност на позицији i различита од -1 и ако јесте, враћамо је. У супротном настављамо извршавање рекурзивне функције. Пре него што функција врати резултат уписујемо га у низ на место i.

```
#include <iostream>
#include <string>
#include <vector>
```

```
using namespace std;

typedef unsigned long long count_t;

count_t brojNacinaDekodiranja_(const string& s, int i, vector<count_t>& memo) {
  if (memo[i] != -1)
    return memo[i];

  int n = s.length();
  if (i == n)
    return memo[n] = 1;
  count_t res = 0;
  if (s[i] != '0')
    res += brojNacinaDekodiranja_(s, i + 1, memo);
  if (i + 1 < n && s[i] != '0' && 10*(s[i] - '0') + s[i + 1] - '0' <= 26)
    res += brojNacinaDekodiranja_(s, i + 2, memo);
  return memo[i] = res;
}

count_t brojNacinaDekodiranja(const string& s) {
  vector<count_t> memo(s.length() + 1, -1);
  return brojNacinaDekodiranja_(s, 0, memo);
}

int main() {
  string s;
  cin >> s;
  cout << brojNacinaDekodiranja(s) << endl;
  return 0;
}
```

Динамичко програмирање навише

Нека dp_i представља број начина да се декодира суфикс речи s који почиње на позицији i (укључујући и њу). С обзиром на структуру рекурзије, низ dp_i можемо одредити сдесна налево.

dp_n представља број начина да се декодира празан суфикс, а то је 1. У супротном у низу имамо бар једну цифру.

Ако је s_i = `'0'`, тада је $dp_i = 0$ (тада ни први, ни други начин декодирања нису могући).

У супротном, треба да размотримо могућност декодирања на први и на други начин. Први начин је увек применљив (јер суфикс почиње бар једном цифром s_i за коју смо утврдили да није цифра `'0'`) и у том случају низ можемо декодирати на dp_{i+1} начина. Други начин је применљив ако имамо бар две цифре (ако је $i < n - 1$) и ако је број добијен од цифара $s_i s_{i+1}$ (за s_i смо утврдили да није цифра `'0'`) између 1 и 26 и у том случају је број начина добијен на овај начин једнак dp_{i+2}. Једноставности ради dp_{n-1} је могуће одредити посебно, да се не би стално проверавало да ли је $i < n - 1$.

Дакле, $dp_n = 1$, $dp_{n-1} = 0$ ако је s_{n-1} = `'0'`, $dp_{n-1} = 1$ ако је $s_{n-1} \neq$ `'0'`. За све вредности i од $n - 2$ унатраг важи да је $dp_i = 0$ ако је s_i = `'0'`, да је $dp_i = dp_{i+1}$, ако $s_i s_{i+1}$ дају број који није између 1 и 26, тј. $dp_i = dp_{i+1} + dp_{i+2}$ ако $s_i s_{i+1}$ дају број који јесте између 1 и 26.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

typedef unsigned long long count_t;
```

```
count_t brojNacinaDekodiranja(const string& s) {
  // dp[i] predstavlja broj nacina da se dekodira sufiks reci s koji
  // pocinje na poziciji i (ukljucujuci i nju).
  int n = s.length();
  vector<count_t> dp(n+1);
  // postoji jedan nacin da se dekodira prazan sufiks
  dp[n] = 1;
  // jednocifreni sufiks se moze dekodirati na jedan nacin ako ta
  // cifra nije '0', a na nula nacina inace
  dp[n-1] = s[n-1] != '0';
  // Niz dp rekonstruisemo unatrag
  for (int i = n-2; i >= 0; i--) {
    // Ako je s[i] karakter '0', onda je dp[i] = 0
    if (s[i] == '0')
      dp[i] = 0;
    else {
      dp[i] = dp[i + 1];
      // A sadrzi i dp[i+2], ako karakteri s[i]s[i+1] cine ispravan
      // kod tj. kod <= 26
      if (10 * (s[i]-'0') + (s[i+1]-'0') <= 26)
        dp[i] += dp[i + 2];
    }
  }
  return dp[0];
}

int main() {
  string s;
  cin >> s;
  cout << brojNacinaDekodiranja(s) << endl;
  return 0;
}
```

Приметимо да је за одређивање сваке претходне вредности низа dp потребно знати само њене две наредне вредности. Стога није неопходно чувати цео низ, већ је у сваком тренутку потребно познавати само две његове узастопне вредности (кренувши од последње и претпоследње).

```
#include <iostream>
#include <string>

using namespace std;

typedef unsigned long long count_t;

count_t brojNacinaDekodiranja(const string& s) {
  int n = s.length();
  // Krecemo od dp[n] = 1
  count_t dp2 = 1;
  // i od dp[n-1] sto je 1 ako s[n-1] nije karakter '0' i 0 u suprotnom */
  count_t dp1 = s[n-1] != '0';
  // Niz dp rekonstruisemo unatrag
  for (int i = n-2; i >= 0; i--) {
    // Vrednost dp[i]
    count_t dp;
    // Ako je s[i] karakter '0', onda je dp[i] = 0
    if (s[i] == '0')
      dp = 0;
    else {
      // dp[i] svakako sadrzi dp[i+1]
```

```
      dp = dp1;
      // A sadrzi i dp[i+2], ako karakteri s[i]s[i+1] cine ispravan
      // kod tj. kod <= 26
      if (10 * (s[i]-'0') + (s[i+1]-'0') <= 26)
        dp += dp2;
    }
    // Azuriramo dva poslednja poznata clana niza
    dp2 = dp1;
    dp1 = dp;
  }
  return dp1;
}

int main() {
  string s;
  cin >> s;
  cout << brojNacinaDekodiranja(s) << endl;
  return 0;
}
```

Задатак: Примитивни бинарни записи

Бинарни записи се састоје само од нула и јединица. За бинарни запис ћемо рећи да је примитиван ако се не може представити надовезивањем неколико копија неког краћег бинарног записа (на пример, запис 111 није примитиван јер се састоји од три копије записа 1, запис 101101 није примитиван јер се састоји од две копије записа 101, док је запис 11001 примитиван). Напиши програм који одређује број различитих примитивних записа дате дужине.

Улаз: Са стандардног улаза се уноси дужина n ($1 \leq n \leq 60$).

Излаз: На стандардни излаз исписати један број који представља број примитивних бинарних записа дужине n (за дато ограничење броја n тај број не прелази опсег 64-битног целобројног типа).

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
4	12	10	990	50	1125899873287200

Решење

Рекурзивно решење

Оба записа дужине 1 (и запис 0 и запис 1) су примитивни. Сваки запис дужине $n > 1$ који није примитиван може се разложити на понављање краћих примитивних записа (ако се разложи на понављање записа који нису примитивни, тај запис се може разложити на понављање мањих, примитивних). Приметимо да примитивни записи на неки начин одговарају простим бројевима (сложени бројеви се увек могу разложити на комбинацију простих). Када се сложени запис (запис који није примитиван) дужине n разложи на понављање неког примитивног, дужина тог примитивног записа мора да дели n. Сваки примитивни запис који је дужине која дели број n понављањем даје неки сложени запис дужине n и различити такви примитивни записи дају различите сложене записе дужине n. Сложених записа дужине $n > 1$ дакле има онолико колико је збир свих примитивних записа за све дужине d које деле број n. Ако са $p(n)$ обележимо број примитивних записа дужине n тада је број сложених записа дужине n једнак $\sum_{d|n,d<n} p(d)$. Пошто свих записа дужине n има 2^n, за број примитивних записа дужине $n > 1$ важи да је $p(n) = 2^n - \sum_{d|n,d<n} p(d)$. Ово даје рекурентну везу на основу које можемо израчунати тражени број примитивних записа.

```
#include <iostream>

using namespace std;

long long broj_primitivnih(int n) {
  if (n == 1) return 2;

  long long broj = broj_primitivnih(1);
```

```
  int d;
  for (d = 2; d*d < n; d++)
    if (n % d == 0)
      broj += broj_primitivnih(d) + broj_primitivnih(n/d);
  if (d*d == n)
    broj += broj_primitivnih(d);
  return (1ll << n) - broj;
}

int main() {
  int n;
  cin >> n;
  cout << broj_primitivnih(n) << endl;
  return 0;
}
```

Динамичко програмирање

Рекурзија је таква да се исти рекурзивни позиви понављају више пута. На пример, када рачунамо број примитивних записа дужине 12, рачунаћемо број примитивних дужине 2, али и број примитивних записа дужине 6, што ће поново проузроковати израчунавање броја примитивних записа дужине 2. С обзиром да су допуштене вредности улазних параметара мале, ово не представља озбиљан проблем. Једно решење је да ово решимо мемоизацијом.

```
#include <iostream>
#include <vector>

using namespace std;

long long broj_primitivnih_(int n, vector<long long>& memo) {
  if (memo[n] > 0)
    return memo[n];

  if (n == 1) return memo[n] = 2;

  long long broj = broj_primitivnih_(1, memo);
  int d;
  for (d = 2; d*d < n; d++)
    if (n % d == 0)
      broj += broj_primitivnih_(d, memo) + broj_primitivnih_(n/d, memo);
  if (d*d == n)
    broj += broj_primitivnih_(d, memo);
  return memo[n] = (1ll << n) - broj;
}

long long broj_primitivnih(int n) {
  vector<long long> memo(n + 1, 0);
  return broj_primitivnih_(n, memo);
}


int main() {
  int n;
  cin >> n;
  cout << broj_primitivnih(n) << endl;
  return 0;
}
```

Проблем преклапајућих рекурзивних позива можемо решити и динамичким програмирањем навише.

```cpp
#include <iostream>
#include <vector>

using namespace std;

long long broj_primitivnih(int N) {
  vector<long long> dp(N + 1, 0);
  dp[1] = 2;
  for (int n = 2; n <= N; n++) {
    long long broj = dp[1];
    int d;
    for (d = 2; d*d < n; d++)
      if (n % d == 0)
        broj += dp[d] + dp[n/d];
    if (d*d == n)
      broj += dp[d];
    dp[n] = (1ll << n) - broj;
  }
  return dp[N];
}


int main() {
  int n;
  cin >> n;
  cout << broj_primitivnih(n) << endl;
  return 0;
}
```

Мебијусова функција

Размотримо још једно решење засновано на теорији бројева. Важи да је $\sum_{d|n} p(d) = 2^n$. На пример, важи да је $2^{12} = p(12) + p(6) + p(4) + p(3) + p(2) + p(1)$ (имамо 2^{12} записа дужине 12, од чега је $p(12)$ примитивних, $p(6)$ се добијају надовезивањем две копије примитивних записа дужине 6, $p(4)$ се добијају надовезивањем три копије примитивних записа дужине 4 итд.). Једнакост важи и за $n = 1$. У овој једнакости непозната је функција p.

Мебијусова формула инверзије каже да ако су f и g две аритметичке функције такве да важи $g(n) = \sum_{d|n} f(d)$ за свако $n \geq 1$, тада за свако $n \geq 1$ важи да је $f(d) = \sum_{d|n} \mu(d) g\left(\frac{n}{d}\right)$ где је μ Мебијусова функција дефинисана тако да је $\mu(n) = 1$ ако су сви прости фактори броја n различити и има их паран број, $\mu(n) = -1$ ако су сви прости фактори броја n различити и има их непаран број и $\mu(n) = 0$ ако n има неки вишеструки прост фактор. Мебијусова функција је мултипликативна (важи $\mu(a \cdot b) = \mu(a)\mu(b)$ за узајамно просте бројеве a и b) и то се може искористити за њено израчунавање (слично као што смо израчунавали збир делилаца Савршени бројеви или Ојлерову функцију Ојлерова функција, коришћењем растављања броја на просте чиниоце Растављање на просте чиниоце).

На основу Мебијусове формуле важи да је $p(n) = \sum_{d|n} \mu\left(\frac{n}{d}\right) 2^d$.

```cpp
#include <iostream>

using namespace std;

int mobius(long long n) {
  if (n == 1) return 1;

  long long d = 2;
  bool paran_broj_faktora = true;
  while (d * d < n) {
    if (n % d == 0) {
      n /= d;
```

```
      if (n % d == 0) return 0; // d je visestruki faktor
      paran_broj_faktora = !paran_broj_faktora;
    }
    d++;
  }

  if (d * d == n)
    return 0;

  return paran_broj_faktora ? -1 : 1;
}

long long broj_primitivnih(int n) {
  long long broj = 0;
  int d;
  for (d = 1; d * d < n; d++) {
    if (n % d == 0) {
      broj += mobius(d) * (1ll << (n / d));
      broj += mobius(n / d) * (1ll << d);
    }
  }
  if (d * d == n)
    broj += mobius(d) * (1ll << d);
  return broj;
}

int main() {
  int n;
  cin >> n;
  cout << broj_primitivnih(n) << endl;
  return 0;
}
```

Задатак: Број појављивања подниске

Написати програм којим се за две дате ниске x и y, одређује број појављивања ниске y као подниза ниске x. Ниска y је подниз ниске x ако се може добити од ниске x брисањем произвољног броја карактера.

Улаз: Прва линија стандардног улаза садржи ниску x, а друга линија ниску y. Дужине ниски су највише 100 карактера.

Излаз: На стандардном изласу приказати само број појављивања ниске y као подниза ниске x.

Пример

Улаз	*Излаз*
abcbca	3
abc	

Објашњење:

Позиције појављивања подниза су обележене великим словима.

```
ABCbca
ABcbCa
AbcBCa
```

Решење

Размотримо прво рекурзивно решење. Ако су обе ниске непразне, тада се може упоредити њихово последње слово. Ако су им последња слова различита онда је укупан број појављивања подниза једнак броју појављивања у префиксу прве ниске без последњег карактера, а ако су им последња слова једнака, онда је укупан број појављивања једнак том броју увећаном за број појављивања префикса друге ниске без последњег карактера

у префиксу прве ниске без последњег карактера. Базу чине случајеви када је нека ниска празна. Ако је друга ниска празна тада се онда једном јавља као подниз прве ниске, а ако је прва ниска празна (а друга није) тада она не садржи подниз који одговара другој ниски. Пошто се кроз рекурзију стално разматрају префикси ниски, ниске не морамо мењати кроз рекурзију, већ можемо прослеђивати само дужине префикса (нека је n дужина префикса прве, а m дужина префикса друге ниске). Ако је $f(n, m)$ тражени број појављивања, тада важи:

$$
\begin{aligned}
&f(n,0) = 1 \\
&f(0,m) = 0, \quad \text{za } m > 0 \\
&f(n,m) = f(n-1,m) + f(n-1,m-1), \quad \text{за } n,m > 0 \;\; a_n = a_m \\
&f(n,m) = f(n-1,m), \quad \text{za } n,m > 0 \;\; a_n \neq a_m
\end{aligned}
$$

На основу овога је веома једноставно направити рекурзивну имплементацију.

```
#include <iostream>
#include <string>

using namespace std;

long long brojPojavljivanjaPodniza(const string& niz, int duzina_niza,
                                   const string& podniz, int duzina_podniza) {
  if (duzina_podniza == 0)
    return 1;
  if (duzina_niza == 0)
    return 0;
  long long broj = brojPojavljivanjaPodniza(niz, duzina_niza-1,
                                            podniz, duzina_podniza);
  if (niz[duzina_niza-1] == podniz[duzina_podniza-1])
    broj += brojPojavljivanjaPodniza(niz, duzina_niza-1,
                                     podniz, duzina_podniza-1);
  return broj;
}

long long brojPojavljivanjaPodniza(const string& niz,
                                   const string& podniz) {
  return brojPojavljivanjaPodniza(niz, niz.length(), podniz, podniz.length());
}

int main () {
  string niz, podniz;
  cin >> niz >> podniz;
  cout << brojPojavljivanjaPodniza(niz, podniz) << endl;
  return 0;
}
```

С обзиром на то да ће се у наивној рекурзивној имплементацији исти рекурзивни позиви извршавати више пута, таква имплементација биће неефикасна. Ефикасност се лако може поправити техником динамичког програмирања (било мемоизацијом, било динамичким програмирањем навише). Пошто функција има два параметра, вредности рекурзивних позива можемо памтити у матрици.

Прикажимо ову матрицу на наредном примеру.

```
abcbca
abc

    0  1  2  3
  | -----------
0 | 1  0  0  0
1 | 1  1  0  0
```

```
2 | 1  1  1  0
3 | 1  1  1  1
4 | 1  1  2  1
5 | 1  1  2  3
6 | 1  2  2  3
```

Пошто елемент на позицији (n, m) зависи од елемената на позицијама $(n-1, m)$ и $(n-1, m-1)$ матрицу можемо попуњавати било врсту по врсту, било колону по колону.

Ако претпоставимо да попуњавамо врсту по врсту матрице можемо извршити меморијску оптимизацију тако што само чувамо елементе једне врсте. У првој врсти иницијализујемо све елементе на нулу, осим првог који иницијализујемо на јединицу. Приликом преласка са текуће на наредну врсту, ажурирање можемо вршити с десна на лево. Ако су одговарајући карактери ниски једнаки, тада текући елемент увећавамо за њему претходни.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

long long brojPojavljivanjaPodniza(string niz, string podniz) {
  int n = niz.length();
  int m = podniz.length();
  vector<long long> dp(m + 1, 0);
  dp[0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = m; j >= 1; j--)
      if (niz[i-1] == podniz[j-1])
        dp[j] += dp[j-1];
  return dp[m];
}

int main() {
  string niz, podniz;
  cin >> niz >> podniz;
  cout << brojPojavljivanjaPodniza(niz, podniz) << endl;
  return 0;
}
```

7.2 Оптимизација коришћењем динамичког програмирања

Задатак: Едит растојање

Едит-растојање између две ниске се дефинише у терминима операција уметања, брисања и измена слова прве речи којима се може добити друга реч. Свака од ове три операције има своју цену. Дефинисати програм који израчунава најмању цену операција којима се од прве ниске може добити друга. На пример, ако је цена сваке операције јединична, тада се ниска `zdravo` може претворити у `bravo!` најефикасније операцијом измене слова `z` у `b`, брисања слова `d` и уметања карактера `!`.

Улаз: Са стандардног улаза се учитавају две ниске дужине највише 100 карактера, а затим цене операције уметања, брисања и измене (природни бројеви од 1 до 10, сваки у посебном реду).

Излаз: На стандардни излаз исписати тражену вредност едит-растојања.

Пример 1

Улаз	*Излаз*
zdravo bravo! 1 1 1	3

Пример 2

Улаз	*Излаз*
kitten sitting 1 2 3	7

Решење

Изведимо прво индуктивно-рекурзивну конструкцију.

- Ако је прва ниска празна, најефикаснији начин да се од ње добије друга ниска је да се уметне један по један карактер друге ниске, тако да је минимална цена једнака производу цене операције уметања и броја карактера друге ниске.
- Ако је друга ниска празна, најефикаснији начин да се од прве ниске добије празна је да се један по један њен карактер избрише, тако да је минимална цена једнака производу цене операције брисања и броја карактера прве ниске.
- Индуктивна хипотеза ће бити да умемо да решимо проблем за било која два префикса прве и друге ниске. Ако су последња слова прве и друге ниске једнака, онда је потребно претворити префикс без последњег слова прве ниске у префикс без последњег слова друге ниске. Ако нису, онда имамо три могућности. Једна је да изменимо један од та два карактера у онај други и онда да, као у претходном случају, преведемо префиксе без последњих карактера један у други. Друга могућност је да обришемо последњи карактер прве ниске и пробамо да претворимо тако њен добијени префикс у другу ниску. Трећа могућност је да прву ниску трансформишемо у префикс друге ниске без последњег карактера и да затим додамо последњи карактер друге ниске.

На основу овога лако можемо дефинисати рекурзивну функцију која израчунава едит-растојање. Да нам се ниске не би мењале током рекурзије (што може бити споро), ефикасније је да ниске прослеђујемо у неизмењеном облику и да само прослеђујемо бројеве карактера њихових префикса који се тренутно разматрају.

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int cenaUmetanja, cenaBrisanja, cenaIzmene;

int editRastojanje(const string& s1, const string& s2, int n1, int n2) {
  if (n1 == 0)
    return n2 * cenaUmetanja;
  if (n2 == 0)
    return n1 * cenaBrisanja;
  if (s1[n1-1] == s2[n2-1])
    return editRastojanje(s1, s2, n1-1, n2-1);
  int r1 = editRastojanje(s1, s2, n1-1, n2) + cenaUmetanja;
  int r2 = editRastojanje(s1, s2, n1, n2-1) + cenaBrisanja;
  int r3 = editRastojanje(s1, s2, n1-1, n2-1) + cenaIzmene;
  return min({r1, r2, r3});
}

int editRastojanje(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  return editRastojanje(s1, s2, n1, n2);
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
```

```
  cin >> cenaUmetanja >> cenaBrisanja >> cenaIzmene;
  cout << editRastojanje(s1, s2) << endl;
  return 0;
}
```

Решење директном рекурзијом је, наравно, неефикасно због преклапајућих рекурзивних позива. Алгоритам динамичког програмирања за овај проблем познат је под именом Вагнер-Фишеров алгоритам. Резултате за префиксе дужине i и j памтићемо у матрици на пољу (i, j). Дакле, ако су дужине ниски n_1 и n_2, потребна нам је матрица димензије $(n_1+1) \times (n_2+1)$, а коначан резултат ће се налазити на месту (n_1, n_2). У наставку ћемо приказати решење помоћу динамичког програмирања навише. Ако матрицу попуњавамо врсту по врсту, слева надесно, приликом израчунавања елемента на позицији (i, j), биће израчунати сви елементи матрице од којег он зависи (а то су $(i-1, j-1)$, $(i-1, j)$ и $(i, j-1)$).

Под претпоставком да су цене јединичне, за ниске `zdravo` и `bravo!` добија се следећа матрица.

```
    b r a v o !
  0 1 2 3 4 5 6
  -------------
 0|0 1 2 3 4 5 6
z1|1 1 2 3 4 5 6
d2|2 2 2 3 4 5 6
r3|3 3 2 3 4 5 6
a4|4 4 3 2 3 4 5
v5|5 5 4 3 2 3 4
o6|6 6 5 4 3 2 3
```

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int editRastojanje(const string& s1, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2+1);

  dp[0][0] = 0;
  for (int i = 0; i <= n1; i++)
    dp[i][0] = i * cenaBrisanja;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      if (s1[i-1] == s2[j-1])
        dp[i][j] = dp[i-1][j-1];
      else {
        int r1 = dp[i-1][j] + cenaUmetanja;
        int r2 = dp[i][j-1] + cenaBrisanja;
        int r3 = dp[i-1][j-1] + cenaIzmene;
        dp[i][j] = min({r1, r2, r3});
      }
    }

  return dp[n1][n2];
}
```

```
int main() {
  string s1, s2;
  cin >> s1 >> s2;
  int cenaUmetanja, cenaBrisanja, cenaIzmene;
  cin >> cenaUmetanja >> cenaBrisanja >> cenaIzmene;
  cout << editRastojanje(s1, s2, cenaUmetanja, cenaBrisanja, cenaIzmene) << endl;
  return 0;
}
```

Овде нам нису битне саме измене, већ само растојање (на пример, ако се врши провера да ли су две ниске блиске приликом претраге у којој се допушта да је корисник направио и неколико словних грешака). Пошто елементи текућег реда зависе само од претходног, можемо извршити меморијску оптимизацију и истовремено чувати само један ред. Током ажурирања елемента на позицији j његов део на позицијама строго мањим од j ће чувати елементе текућег реда i, део од позиције j надаље ће чувати елементе претходног реда $i-1$. Променљива `prethodni` ће чувати вредност са поља $(i-1, j-1)$, а променљива `tekuci` ће чувати вредност са поља $(i-1, j)$.

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int editRastojanje(const string& s1, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1);
  dp[0] = 0;
  for (int j = 0; j <= n2; j++)
    dp[j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    dp[0] = i * cenaBrisanja;
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni;
      else {
        int r1 = tekuci + cenaUmetanja;
        int r2 = dp[j-1] + cenaBrisanja;
        int r3 = prethodni + cenaIzmene;
        dp[j] = min({r1, r2, r3});
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
  int cenaUmetanja, cenaBrisanja, cenaIzmene;
  cin >> cenaUmetanja >> cenaBrisanja >> cenaIzmene;
  cout << editRastojanje(s1, s2, cenaUmetanja, cenaBrisanja, cenaIzmene) << endl;
  return 0;
}
```

Задатак: Најдужи заједнички подниз две ниске

Напиши програм који израчунава дужину највећег заједничког подниза две ниске. Подниз чине карактери ниске који не морају бити узастопни, али се јављају у истом редоследу као у оригиналној ниски. На пример за ниске `abacbc` и `babbca` најдужа заједничка подниска је `babc`.

Улаз: Две линије стандардног улаза садрже две ниске које се састоје од малих слова енглеске азбуке и дугачке су највише 1000 карактера.

Излаз: На стандардни излаз исписати само тражену дужину.

Пример

Улаз	*Излаз*
`xmjyauz` `mzjawxu`	`4`

Решење

Ако је било која од две ниске празна, тада је једини њен подниз празан, па је дужина најдужег заједничког подниза једнака нули. Ако су обе ниске непразне, тада можемо упоредити њихова последња слова. Ако су она једнака, могу бити укључена у најдужи заједнички подниз и проблем се рекурзивно своди на проналажење најдужег заједничког подниза њихових префикса. У супротном, није могуће да оба последња слова буду укључена у заједнички подниз. Зато разматрамо најдужи заједнички подниз прве ниске и префикса друге ниске без њеног последњег слова и заједнички подниз друге ниске и префикса прве ниске без њеног последњег слова. Дужи од два подниза биће најдужи заједнички подниз те две ниске. Нагласимо и да није неопходно разматрати најдужи заједнички подниз два префикса, јер се проширивањем неког од два префикса за последње слово не може добити подниз који би био краћи. Дакле, пошто рекурзија тече по префиксима ниски, једини променљиви параметри током рекурзије могу бити дужине тих префикса.

Ако са $f(m, n)$ означимо дужину најдужег заједничког подниза префикса ниске a дужине m и префикса ниске b дужине n, тада важи следећа рекурентна веза:

$$
\begin{aligned}
f(0, n) &= 0 \\
f(m, 0) &= 0 \\
f(m, n) &= f(m-1, n-1) + 1, \quad \text{za } a_{m-1} = b_{n-1} \\
f(m, n) &= \max\left(f(m, n-1), f(m-1, n)\right), \quad \text{za } a_{m-1} \neq b_{n-1}
\end{aligned}
$$

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2) {
    if (n1 == 0 || n2 == 0)
       return 0;
    int rez = max(najduziZajednickiPodniz(s1, n1, s2, n2-1),
                  najduziZajednickiPodniz(s1, n1-1, s2, n2));
    if (s1[n1-1] == s2[n2-1])
       rez = max(rez, najduziZajednickiPodniz(s1, n1-1, s2, n2-1) + 1);
    return rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  return najduziZajednickiPodniz(s1, s1.size(), s2, s2.size());
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
```

```
  cout << najduziZajednickiPodniz(s1, s2) << endl;
  return 0;
}
```

У директном рекурзивном решењу има много преклапајућих рекурзивних позива. Стога је ефикасност могуће поправити техником динамичког програмирања. Један могући приступ је да употребимо мемоизацију. Вредност дужине најдужег подниза за сваки пар дужина префикса можемо памтити у матрици.

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2,
                            vector<vector<int>>& memo) {
  if (memo[n1][n2] != -1)
    return memo[n1][n2];

  if (n1 == 0 || n2 == 0)
    return memo[n1][n2] = 0;
  int rez = max(najduziZajednickiPodniz(s1, n1, s2, n2-1, memo),
                najduziZajednickiPodniz(s1, n1-1, s2, n2, memo));
  if (s1[n1-1] == s2[n2-1])
    rez = max(rez, najduziZajednickiPodniz(s1, n1-1, s2, n2-1, memo) + 1);
  return memo[n1][n2] = rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> memo(n1+1);
  for (int i = 0; i <= n1; i++)
    memo[i].resize(n2 + 1, -1);

  return najduziZajednickiPodniz(s1, n1, s2, n2, memo);
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
  cout << najduziZajednickiPodniz(s1, s2) << endl;
  return 0;
}
```

Проблем прекалпајућих рекурзивних позива се може решити ако се употреби динамичко програмирање навише. Дужине најдужих поднизова префикса можемо чувати у матрици. Елемент матрице на позицији (m, n) зависи само од елемената на позицијама $(m-1, n)$, $(m, n-1)$ и $(m-1, n-1)$, тако да матрицу пожемо да попуњавамо било врсту по врсту, било колону по колону. Прикажимо матрицу за пример две ниске.

```
xmjyauz
mzjawxu

     mzjawxu
    01234567
   ---------
  0|00000000
x 1|00000011
m 2|01111111
j 3|01122222
y 4|01122222
```

```
a 5|01123333
u 6|01123334
z 7|01223334
```

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2 + 1);

  for (int i = 0; i <= n1; i++)
    dp[i][0] = 0;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = 0;

  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
      if (s1[i-1] == s2[j-1])
        dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
    }

  return dp[n1][n2];
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
  cout << najduziZajednickiPodniz(s1, s2) << endl;
  return 0;
}
```

Можемо приметити да се приликом попуњавања матрице врсту по врсту садржај сваке наредне врсте попуњава само на основу претходне врсте. Стога није потребно истовремено памтити целу матрицу, већ је довољно памтити само једну, текућу врсту. Ажурирање врсте морамо вршити с лева надесно, јер сваки елемент у текућој врсти зависи од елемента који му претходи у тој врсти. Приметимо да нам је у неком тренутку потребно да знамо претходни елемент текуће врсте, а понекад претходни елемент претходне врсте, тако да приликом ажурирања врсте морамо да у помоћној променљивој памтимо стару вредност претходног елемента врста (јер се ажурирањем претходног елемента његова стара вредност губи, а она нам може затребати у случају да су одговарајући карактери у нискама једнаки).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
```

```
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni + 1;
      else
        dp[j] = max(dp[j-1], dp[j]);
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
  cout << najduziZajednickiPodniz(s1, s2) << endl;
  return 0;
}
```

Задатак: Најдужи подниз који се понавља

Написати програм којим се у датом стрингу s налази дужина n најдужег подниза (не обавезно узастпоних карактера), који се у стрингу s понавља, али тако да за свако $0 \leq i < n$, i-ти карактери поднизова немају исту позицију у стрингу s.

Улаз: Прва линија стандардног улаза садржи стринг s дужине највише 1000.

Излаз: На стандардном излазу приказати у једној линији тражену дужину.

Пример

Улаз	*Излаз*
`mahaabkbc`	`3`

Објашњење

Најдужи подниз који се понаља по траженим условима је `aab`. Једно појављивање добија се кад се из стринга s узимају редом елементи са позиција 1, 3 и 5, а друго појављивање добијамо ако редом узимамо елементе са позиција 3, 4 и 7. Елемент са позиције 3 користи се у оба подниза али на различитим позицијама.

Решење

Задатак је веома сличан и решава се на сличан начин задатку Најдужи заједнички подниз две ниске. Иако на први поглед делује да би се задатак могао директно решити свођењем на тај задатак, постоји додатни услов је тај да се слово са исте позиције не може употребити два пута на истој позицији, о коме у решењу морамо водити рачуна.

Задатак се може решити дефинисањем рекурзивне функције која одређује дужину најдужег подниза који се понавља у два префикса дате ниске (функција прима ниску и дужину њена два префикса). Ако је било који префикс празан, излазимо из рекурзије тако што закључујемо да не постоји заједнички подниз између два префикса. У супротном анализирамо да ли су два завршна слова префикса једнака ако јесу, она би се могла узети за последња два слова два појављивања заједничке подниске, међутим, то не сме да се ради ако су префикси исте дужине (јер се тада исто слово понавља на истој позицији унутар подниске). Дакле, ако се префикси завршавају истим словом и ако су различите дужине, рекурзивно одређујемо резултат за префиксе који се добијијају уклањањем последњег слова из једног и из другог префикса и враћамо дужину њиховог заједничког подниза увећану за 1 (јер подниз проширујемо заједничким завршним словом оба префикса). У супротном знамо да бар једно од последњих слова префикса не учествује у поднизу, па анализирамо две могућности (изостављање последњег слова првог и последњег слова другог префикса) и узимамо бољу од њих.

```
#include <iostream>

using namespace std;

int podniz(const string& s, int n1, int n2) {
  if (n1 == 0 || n2 == 0)
```

```
    return 0;
  if (n1 != n2 && s[n1-1] == s[n2-1])
    return podniz(s, n1-1, n2-1) + 1;
  return max(podniz(s, n1-1, n2),
             podniz(s, n1, n2-1));
}

int podniz(const string& s) {
  return podniz(s, s.length(), s.length());
}

int main() {
  string s;
  cin >> s;
  cout << podniz(s) << endl;
}
```

У рекурзивном решењу долази до преклапања рекурзивних позива, па је потребно употребити неку технику динамичког програмирања. Рекурзивно решење можемо оптимизовати применом мемоизације. Пошто функција има два променљива параметра (дужине префикса) за мемоизацију употребљавамо матрицу.

```
#include <iostream>
#include <vector>

using namespace std;

int podniz(const string& s, int n1, int n2,
                                vector<vector<int>>& memo) {
  if (memo[n1][n2] != -1)
    return memo[n1][n2];

  if (n1 == 0 || n2 == 0)
    return memo[n1][n2] = 0;

  if (n1 != n2 && s[n1-1] == s[n2-1])
    return memo[n1][n2] = podniz(s, n1-1, n2-1, memo) + 1;
  return memo[n1][n2] = max(podniz(s, n1-1, n2, memo),
                            podniz(s, n1, n2-1, memo));
}

int podniz(const string& s) {
  int n = s.length();
  vector<vector<int>> memo(n+1, vector<int>(n+1, -1));
  return podniz(s, n, n, memo);
}

int main() {
  string s;
  cin >> s;
  cout << podniz(s) << endl;
}
```

Задатак је могуће решити и динамичким програмирањем навише, тако што ћемо матрицу попунити врсту по врсту (елементе прве врсте и прве колоне који одговарају празним префиксима попуњавамо нулама, а елемент на позицији (i, j) попуњавамо на основу вредности $(i-1, i-1)$ или вредности $(i, j-1)$ и $(i-1, j)$).

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
```

```
using namespace std;

int podniz(const string& s) {
  int n = s.length();
  vector<vector<int>> dp(n+1, vector<int>(n+1));
  for (int i = 0; i <= n; i++) {
    dp[i][0] = 0;
    dp[0][i] = 0;
  }
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      if (s[i-1] == s[j-1] && i != j)
        dp[i][j] = dp[i-1][j-1] + 1;
      else
        dp[i][j] = max(dp[i][j-1], dp[i-1][j]);

  return dp[n][n];
}

int main() {
  string s;
  cin >> s;
  cout << podniz(s) << endl;
  return 0;
}
```

Задатак: Најдужа заједничка подниска

Дефинисати функцију која одређује дужину најдуже заједничке подниске (узастопних карактера) две дате ниске. На пример, за ниске `ababc` и `babbca` најдужа заједничка подниска је `ab`.

Улаз: Свака од две линије стандардног улаза садржи ниску састављену од малих слова енглеске абецеде, дужине највише 1000 карактера.

Излаз: На стандардни излаз исписати дужину најдуже заједничке подниске.

Пример

Улаз	*Излаз*
`acabacamaca` `macabacaaca`	7

Задатак: Најдужа подниска који се понавља без преклапања

Написати програм којим се у датом стрингу s налази дужина најдужег подстринга који се у стрингу s понавља, али тако да се подстрингови који се понављају не преклапају (немају ни један елемент са исте позиције стринга s).

Напоменимо да подстринг чине узастопни елементи стринга.

Улаз: Прва линија стандардног улаза садржи стринг s дужине највише 100.

Излаз: На стандардном излазу у једној линији приказати најдужи подстринг стринга s који се понавља а при томе нема преклапања елемената, ако такав подстринг не постоји приказати `-`. Ако постоји више подстрингова исте највеће дужине, приказати први који се појављује у s (гледано с лева надесно).

Пример 1

Улаз	*Излаз*
`banana`	`an`

Пример 2

Улаз	*Излаз*
`abcdabecd`	`ab`

Пример 3

Улаз	*Излаз*
`grad`	`-`

Задатак: Најдужи растући подниз

Напиши програм који одређује најдужи строго растући подниз (не обавезно узастопних елемената) унутар датог низа.

Улаз: Са стандардног улаза се учитава број елемената низа n ($1 \leq n \leq 50000$), а затим елементи низа (цели бројеви, сваки у посебном реду).

Излаз: На стандардни излаз исписати дужину најдужег растућег подниза.

Пример

Улаз	*Излаз*
10 3 2 6 9 5 4 3 7 2 8	4

Објашњење

Један растући подниз дужине 4 је `2 6 7 8`.

Решење

Приказаћемо два решења заснована на динамичком програмирању, која су различите ефикасности.

У првој групи решења разматраћемо позицију по позицију у низу и одредићемо најдужи растући подниз чији је последњи елемент на свакој од њих. Приликом одређивања дужине најдужег растућег подниза који се завршава на позицији $i \geq 0$, претпоставићемо да за сваку претходну позицију (ако их има) умемо да одредимо дужину најдужег растућег подниза који се на њој завршава. Низ који се завршава на позицији i сигурно садржи елемент a_i, а може продужити све оне низове који се завршавају на некој позицији $0 \leq j < i$ ако је $a_j < a_i$. Да би низ који се завршава на позицији j био што дужи, његов префикс који се завршава на позицији j мора бити што дужи (а дужине тих низова можемо одредити рекурзивно). Зато од свих низова који се завршавају на позицијама j таквим да је $a_j < a_i$ одређујемо најдужи и продужавамо га елементом a_i (ако таквих елемената нема, тада је најдужи низ који се завршава на позицији a_i једночлан).

```
#include <iostream>
#include <vector>

using namespace std;

// pronalazi duzinu najduzeg rastuceg podniza niza a koji se zavrsava
// elementom na poziciji i
int najduziRastuciPodniz(const vector<int>& a, int i) {
  int maksI = 1;
  for (int j = 0; j < i; j++)
    if (a[j] < a[i]) {
      int maksJ = najduziRastuciPodniz(a, j);
      if (maksJ + 1 > maksI)
        maksI = maksJ + 1;
    }
  return maksI;
}

int najduziRastuciPodniz(const vector<int>& a) {
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
```

```
    int maksI = najduziRastuciPodniz(a, i);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << najduziRastuciPodniz(a) << endl;
  return 0;
}
```

Када се претходна рекурзивна веза директно имплементира, долази до понављања идентичних рекурзивних позива, што доводи до велике неефикасности. Проблем решавамо динамичким програмирањем. За мемоизацију довољно је да памтимо низ дужина најдужих растућих низова који се завршавају на свакој позицији у низу. Пошто су све те дужине веће или једнаке од 1 (сваки елемент сам за себе чини растући низ), низ у који меморишемо решења можемо иницијализовати нулама (што значи да је тражена дужина још непозната).

```
#include <iostream>
#include <vector>
using namespace std;

// pronalazi duzinu maksP najduzeg rastuceg podniza niza a koji se
// zavrsava elementom na poziciji p, kao i duzinu maks najduzeg
// rastuceg podniza unutar prefiksa koji se zavrsava na poziciji p
int najduziRastuciPodniz(const vector<int>& a, int p,
                         vector<int>& memo) {
  if (memo[p] != 0)
    return memo[p];
  int maksP = 1;
  for (int i = 0; i < p; i++) {
    if (a[i] < a[p]) {
      int maksI = najduziRastuciPodniz(a, i, memo);
      if (maksI + 1 > maksP)
        maksP = maksI + 1;
    }
  }
  return memo[p] = maksP;
}

int najduziRastuciPodniz(const vector<int>& a) {
  vector<int> memo(a.size(), 0);
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
    int maksI = najduziRastuciPodniz(a, i, memo);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}

int main() {
  ios_base::sync_with_stdio(false);
```

```
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << najduziRastuciPodniz(a) << endl;
  return 0;
}
```

Једноставно можемо формулисати и динамичко програмирање навише, тако што низ попуњавамо слева надесно. Након попуњавања низа одређујемо његов максимум. Нагласимо да сваки наредни елемент низа потенцијално зависи од великог броја претходних тако да није могуће редуковати меморијску сложеност тиме што би се памтили само неки елементи низа (морамо увек знати дужине свих претходних елемената). Решење које се добија на овај начин је меморијске сложености $O(n)$ и временске сложености $O(n^2)$.

Прикажимо на примеру како ће се тај низ попуњавати.

```
i  0 1 2 3 4 5 6 7 8 9
ai 3 2 6 9 5 4 3 7 8 2
dp 1 1 2 3 2 2 2 3 4 1
```

На пример, када израчунавамо елемент на позицији 5 анализирамо низове који се завршавају елементима мањим од вредности 4 која се налази на позицији 5. То су вредности 3 на позицији 0 и 2 на позицији 1. У оба случаја максимална дужина подниза који се завршава на тој позицији је 1, па се било који од тих низова продужава елементом 4 и добија се растући низ дужине 2.

```
#include <iostream>
#include <vector>

using namespace std;

int najduzi_rastuci_podniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++)
      if (a[i] > a[j] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;
  }

  int max = dp[0];
  for (int i = 0; i < n; i++)
    if (dp[i] > max)
      max = dp[i];
  return max;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  cout << najduzi_rastuci_podniz(a) << endl;

  return 0;
}
```

Претходно решење је сложености $O(n^2)$, али изменом индуктивно-рекурзивне конструкције можемо добити и много ефикасније решење. Кључна идеја је да претпоставимо да уз дужину d_{max} најдужег растућег подниза до сада обрађеног дела низа можемо за сваку дужину подниза $1 \leq d \leq d_{max}$ да одредимо најмањи елемент којим се завршава неки растући подниз дужине d. Приметимо да низ тих вредности увек строго растући (ако постоји строго растући низ дужине d који се завршава неким елементом a_i, тада се његов префикс дужине $d-1$ мора завршавати неким елементом који је строго мањи од елемента a_i).

Низ обрађујемо елемент по елемент. Размотримо један пример (колоне табеле су означене дужинама низа), а елементи низа који се обрађују су написани десно.

```
1 2 3 4 5 6 7 8 9 10
- - - - - - - - - -
3 - - - - - - - - -         3
2 - - - - - - - - -         2
2 6 - - - - - - - -         6
2 6 9 - - - - - - -         9
2 5 9 - - - - - - -         5
2 4 9 - - - - - - -         4
2 3 9 - - - - - - -         3
2 3 7 - - - - - - -         7
2 3 7 - - - - - - -         2
2 3 7 8 - - - - - -         8
```

Ако се досадашњи најдужи подниз завршавао елементом који је мањи од текућег, онда смо нашли подниз који је дужи за један и најмањи елемент на крају тог подниза је текући. То се у примеру дешава приликом обраде елемента `3`, елемента `6`, елемента `9` и елемента `8`.

Размотримо ситуацију у којој обрађујемо елемент `5`. До тада смо видели елементе `3`, `2`, `6` и `9`. Елемент `2` на првој позицији у табели означава да је најмањи елемент којим се може завршити једночлани растући низ једнак `2`. Елемент `6` на другој позицији у табели означава да је најмањи елемент којим се може завршити двочлани растући низ једнак `6` (у питању је низ `2 6` или низ `3 6`). Елемент `9` на трећој позицији у табели означава да је најмањи елемент којим се може завршити трочлани растући низ једнак `9` (у питању је низ `2 6 9` или `3 6 9`). Пошто је `5` мањи од `9` ниједан од ових трочланих низова није могуће проширити елементом `5`, па је четворочланих растућих низова нема. Поставља се питање да ли се можда трочлани низови могу завршити елементом `5`, но ни то није могуће. Наиме, пошто је у табели двочланим низовима придружена вредност `6`, то значи да се сви двочлани растући низови завршавају бар са `6`, па није могуће `9` заменити са `5`. Са друге стране, пошто је `5` веће од `2`, завршни елемент двочланих низова `6` је могуће заменити са `5` и тиме добити мању завршну вредност двочланих низова (то су у овом случају низови `3 5` и `2 5`). Дакле, у табели вредност `6` треба заменити вредношћу `5`. Вредност `2` лево од `6` нема смисла заменити са `5`, јер би се тиме завршна вредност једночланих низова увећала, а ми у табели памтимо најмање завршне вредности.

На основу анализе овог примера можемо да закључимо да је приликом анализе сваког текућег елемента потребно пронаћи прву позицију d у табели на којој се налази елемент који је већи или једнак од текућег и позицију d уместо тога уписати текући елемент. Ако су сви елементи мањи од текућег (ако је $d = d_{max}$), онда се текући елемент додаје на крај низа (и у том случају заправо радимо исто - уписујемо елемент на позицију d). Остали елементи у табели остају непромењени. Заиста на свим позицијама у табели лево од позиције d уписани су елементи строго мањи од текућег и њиховом заменом са текућим се не би смањила вредност завршног елемената тих низова. За елементе десно од позиције d, иако су већи од текућег, ажурирање није могуће. У свим низовима дужине $d' > d$ неки префикс се морао завршавати елементом на позицији d или елементом већим од њега, а пошто је он био већи или једнак од текућег, заменог последњег елемента текућим не бисмо добили више растући низ.

Кључни добитак настаје када се примети да, пошто су елементи у табели сортирани, позицију првог елемента који је већи или једнак од текућег можемо остварити бинарном претрагом. Отуда следи ефикасна имплементација (у низу `dp` вредност најмањег завршног елемента за низове дужине d памтимо на позицији $d-1$). Временска сложеност такве имплементације је $O(n \cdot \log(n))$, док је меморијска сложеност $O(n)$.

Бинарна претрага може бити извршена библиотечком функцијом.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

int najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  int max = 0;
  for (int i = 0; i < n; i++) {
    auto it = lower_bound(dp.begin(), next(dp.begin(), max), a[i]);
    *it = a[i];
    int d = distance(dp.begin(), it);
    if (d + 1 > max)
      max = d + 1;
  }
  return max;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n, 0);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  cout << najduziRastuciPodniz(a) << endl;

  return 0;
}
```

Још једна могућност је да се бинарна претрага ручно имплементира (исто као у задатку Први већи и последњи мањи).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int prviVeciIliJednak(const vector<int>& a, int l, int d, int x) {
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] < x)
      l = s + 1;
    else
      d = s;
  }
  return l;
}

int  najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n+1);
  int max = 0;
  for (int i = 0; i < n; i++) {
    int k = prviVeciIliJednak(dp, 0, max, a[i]);
    dp[k] = a[i];
    if (k + 1 > max)
      max = k + 1;
```

```
  }
  return max;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  cout << najduziRastuciPodniz(a) << endl;

  return 0;
}
```

Постоји веома једноставно свођење овог проблема на проблем проналажења најдужег заједничког подниза два низа, чије смо решење већ приказали у задатку Најдужи заједнички подниз две ниске. Наиме, дужина најдужег растућег подниза датог низа једнака је дужини најдужег заједничког подниза тог низа и низа који се добија неопадајућим сортирањем и уклањањем дупликата тог низа.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int najduziZajednickiPodniz(const vector<int>& s1, const vector<int>& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1]) {
        dp[j] = prethodni + 1;
      } else {
        dp[j] = max(dp[j-1], dp[j]);
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int najduziRastuciPodniz(const vector<int>& a) {
  // sortirana kopija niza a
  vector<int> b = a;
  sort(begin(b), end(b));
  // uklanjamo duplikate iz vektora b
  b.erase(unique(begin(b), end(b)), end(b));
  return najduziZajednickiPodniz(a, b);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
```

```
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << najduziRastuciPodniz(a) << endl;
  return 0;
}
```

Задатак: Падајуће лоптице

Лоптица се испушта са висине и пада кроз простор испуњен препрекама. Када испод ње нема препрека, она пада директно наниже. Када је препрека испод ње, она се зауставља и тада је можемо гурнути било лево, било десно до ивице препреке, након чега она наставља да пада. Напиши програм који за сваку почетну позицију у највишем реду одређује да ли лоптица може да дође до дна.

Улаз: Са стандардног улаза се учитава матрица карактера. Препреке су означене карактерима `x`, а празнине карактерима `.`.

Излаз: На стандардни излаз исписати низ нула и јединица (за поља са којих лоптица не може пасти до дна исписати 0, а за она са којих може 1).

Пример

Улаз	*Излаз*

`x........x...x`
`xx.x.....xx..x`
`...x....xxx.xx`
`..xxx........x`
`..xxx..xx....x`
`..xxxxxxx.xxxx`
	`01110001101110`

Задатак: Допуна нулама до највећег скаларног производа

Дата су два низа a и b природних бројева величине редом n и m при чему је $n \geq m$. Потребно је додати у низу b на произвољним позицијама $m-n$ нула тако да скаларни производ (сума производа $a_i \cdot b_i$, за $0 \leq i < n$) тог вектора и вектора a има максималну вредност. Написати програм којим се одређује максимална вредност траженог скаларног производа.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 100$), следећа линија садржи n природних бројева раздвојених размаком који представљају редом елементе низа a. Затим се на стандардном улазу у једној линији налази природан број m ($m < n$) а у следећој линији се налази m природних бројева раздвојених размаком, који представљају редом елементе низа b.

Излаз: На стандардном излазу приказати у једној линији максималну вредност траженог скаларног производа.

Пример

Улаз	*Излаз*

`5`
`2 3 1 7 8`
`3`
`7 3 6`
	`90`

Објашњење: Максимални скаларни производ се постиже за низ $b = [0, 7, 0, 3, 6]$.

Задатак: Исплата са најмање новчића

Дате су вредности n врста новчића. Написати програм који одређује минималан број новчића потребних за исплату датог износа S, при чему се може се користити и више новчића исте врсте и сваке врсте новчића има произвољно много. Вредности новчића и износ за исплату дати су у истој валути.

Улаз: Прва линија стандардног улаза задржи природан број S ($S \leq 1000$). Друга линија садржи природан број n ($n \leq 100$), у следећих n линија налазе се природни бројеви који представљају вредности за сваку врсту новчића, свака вредност у посебној линији.

Излаз: На стандардном излазу приказати у једној линији минималан број новчића потребан sa исплату износа S.

Пример 1

Улаз	*Излаз*
7 3 1 3 2	3

Објашњење: исплата за износ 7 са најмање новчића је $3, 3, 1$.

Пример 2

Улаз

```
12
3
1
9
6
```

Излаз

```
2
```

Објашњење: исплата за износ 12 са најмање новчића је $6, 6$.

Решење

Анализа последње врсте новчића

Износ 0 се може наплатити са 0 новчића. У супротном, ако је низ расположивих новчића празан, износ није могуће наплатити. У супротном испитујемо могућност да је у износ укључен новчић последње врсте. Ако није, покушавамо да наплатимо износ без коришћења последње врсте новчића, тј. са префиксном низа новчића без последњег елемента. Ако јесте, тада износ мора бити већи или једнак од новчића последње врсте и тада преостали износ покушавамо да наплатимо поново коришћењем свих врста новчића. Ако са $f(n, s)$ најмањи број новчића да се наплати износ s помоћу новчића који припадају префиксу дужине n полазног низа, тада важи

$$
\begin{aligned}
&f(n, 0) = 0, \\
&f(0, s) = +\infty, \quad \text{за } s > 0 \\
&f(n, s) = f(n-1, s), \quad \text{за } s < v_{n-1} \\
&f(n, s) = \min\left(f(n-1, s), 1 + f(n, s - v_{n-1})\right), \quad \text{за } s \geq v_{n-1}
\end{aligned}
$$

На основу овога је веома једноставно дефинисати рекурзивну функцију која израчунава тражени најмањи број новчића.

```
#include <iostream>
#include <algorithm>

using namespace std;

const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int s) {
  // iznos 0 se plaća sa 0 novčića
  if (s == 0)
    return 0;
```

```
  // ako nema novčića pozitivan iznos nije moguće platiti
  if (n == 0)
    return INF;
  // broj novčića ako se ne uzme ni jedan novčić poslednje vrste
  int br = minBrojNovcica(v, n-1, s);
  // ako je iznos veći od novčića poslednje vrste
  if (s >= v[n-1])
    // gledamo da li je bolje ako se uzme jedan novčić poslednje vrste
    br = min(br, minBrojNovcica(v, n, s - v[n-1]) + 1);
  // vraćamo rezultat
  return br;
}

int main() {
  // učitavamo iznos i niz novčića
  int S;
  cin >> S;
  int n;
  cin >> n;
  int v[MAX_V];
  for(int i = 0; i < n; i++)
    cin >> v[i];
  // izračunavamo i ispisujemo minimum
  int br = minBrojNovcica(v, n, S);
  cout << (br == INF ? -1 : br) << endl;
  return 0;
}
```

Јасно је да у претходној функцији може доћи до понављања истих рекурзивних позива, што се може решити техником динамичког програмирања. Пошто функција има два променљива параметра, могуће је употребити матрицу за мемоизацију.

```
#include <iostream>
#include <algorithm>

using namespace std;

const int MAX_V = 100;
const int MAX_S = 2000;
const int INF = MAX_S + 1;

// matrica koju koristimo za memoizaciju
int memo[MAX_V][MAX_S + 1];

int minBrojNovcica(int v[], int n, int s){
  // ako smo već računali vrednost za (n, s), vraćamo upamćeni rezultat
  if (memo[n][s] != 0)
    return memo[n][s];
  // iznos 0 se naplaćuje sa 0 novčića
  if (s == 0)
    return memo[n][s] = 0;
  // ako nema novčića pozitivan iznos nije moguće platiti
  if (n == 0)
    return memo[n][s] = INF;
  // broj novčića ako se ne uzme ni jedan novčić poslednje vrste
  int br = minBrojNovcica(v, n-1, s);
  // ako je iznos veći od novčića poslednje vrste
  if (s >= v[n-1])
    // gledamo da li je bolje ako se uzme jedan novčić poslednje vrste
```

```
    br = min(br, minBrojNovcica(v, n, s - v[n-1]) + 1);
  // vraćamo rezultat
  return memo[n][s] = br;
}

int main() {
  // učitavamo iznos i niz vrednosti novčića
  int S, N;
  int v[MAX_V];
  cin >> S >> N;
  for (int i = 0; i < N; i++)
    cin >> v[i];
  // izračunavamo i ispisujemo najmanji broj novčića
  int br = minBrojNovcica(v, N, S);
  cout << (br == INF ? -1 : br)  << endl;
  return 0;
}
```

Приликом динамичког програмирања навише, матрицу можемо попуњавати врсту по врсту и тако смањити меморијску сложеност. Временска сложеност овог решења је $O(S \cdot N)$, где је S износ, а N број врста новића, док је меморијска сложеност $O(S)$.

```
#include <iostream>
#include <algorithm>

using namespace std;

const int MAX_V = 100;
const int MAX_S = 2000;
const int INF = MAX_S + 1;


// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int N, int S){
  // najmanji broj novčića potrebnih da se plati svaki iznos od 0 do S
  int dp[MAX_S + 1];

  // analizu pokrećemo sa 0 vrsta novčića

  // iznos 0 se plaća sa 0 novčića
  dp[0] = 0;
  // ostale iznose nije moguće platiti
  for (int s = 1; s <= S; s++)
    dp[s] = INF;

  // uključujemo jednu po jednu vrstu novčića
  for (int n = 1; n <= N; n++) {
    // ažuriramo vrednosti svih iznosa
    for (int s = 0; s <= S; s++) {
      // ako je moguće uključiti novčić tekuće vrste
      if (s >= v[n-1])
        // ažuriramo minimum ako je to potrebno
        dp[s] = min(dp[s], dp[s - v[n-1]] + 1);
    }
  }

  // vraćamo najmanji broj novčića za iznos S
  return dp[S];
```

```
}

int main() {
  int S, N;
  int v[MAX_V];
  cin >> S >> N;
  for (int i = 0; i < N; i++)
    cin >> v[i];
  int br = minBrojNovcica(v, N, S);
  cout << (br == INF ? -1 : br) << endl;
  return 0;
}
```

Анализа свих могућности за последњи новчић

Друго могуће решење разматра све могућности за последњи употребљени новчић. То може бити било који новчић који је мањи од текућег износа који треба наплатити, након чега се преостали износ и даље наплаћује помоћу свих могућих новчића. Ако са $f(s)$ обележимо најмањи број новчића потребних да се наплати износ s, при чему се могу користити сви расположиви новчићи, добијамо следећу рекурентну везу.

$$f(0) = 0,$$
$$f(s) = \min_{v_i \leq s} (1 + f(s - v_i))$$

Ако је износ позитиван, али мањи од свих новчића које имамо на располагању тада је минимум једнак $+\infty$. И у овом случају веома једноставно можемо дефинисати рекурзивну функцију.

```
#include <iostream>
#include <algorithm>
using namespace std;

const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// najmanji broj novčića potreban da se naplati iznos s
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int s) {
  // iznos 0 se naplaćuje sa 0 novčića
  if (s == 0)
    return 0;
  // minimalni broj novčića da se naplati iznos s (pretpostavljamo
  // da iznos nije moguće naplatiti)
  int br = INF;
  // razmatramo sve mogućnosti za poslednji novčić
  for (int i = 0; i < n; i++)
    // proveravamo da li je iznos s moguće naplatiti novčićem i
    if (v[i] <= s)
      // određujemo rekurzivno broj novčića za preostali iznos i
      // ažuriramo minimum ako je to potrebno
      br = min(br, minBrojNovcica(v, n, s-v[i]) + 1);
  // vraćamo rezultat
  return br;
}

int main() {
  // učitavamo iznos i niz novčića
  int N, S;
  int v[MAX_V];
```

```
  cin >> S >> N;
  for(int i = 0; i < N; i++)
    cin >> v[i];
  // izračunavamo i ispisujemo minimum
  int br = minBrojNovcica(v, N, S);
  cout << (br == INF ? -1 : br) << endl;
  return 0;
}
```

Пошто долази до понављања истих рекурзивних позива потребно је да употребимо динамичко програмирање. Пошто је само један параметар променљив, за мемоизацију је довољно само да користимо низ.

```
#include <iostream>
#include <algorithm>
using namespace std;

const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// niz koji koristimo za memoizaciju - inicijalizovan podrazumevano na 0
int memo[MAX_S + 1];

// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int S) {
  // ako smo već izračunali broj novčića za iznos S, vraćamo upamćen rezultat
  if (memo[S] != 0)
    return memo[S];
  // iznos 0 se naplaćuje sa 0 novčića
  if (S == 0)
    return memo[S] = 0;
  // minimalni broj novčića da se naplati iznos s (pretpostavljamo
  // da iznos nije moguće naplatiti)
  int br = INF;
  // razmatramo sve mogućnosti za poslednji novčić
  for (int i = 0; i < n; i++)
    // proveravamo da li je iznos s moguće naplatiti novčićem i
    if (v[i] <= S)
      // određujemo rekurzivno broj novčića za preostali iznos i
      // ažuriramo minimum ako je to potrebno
      br = min(br, minBrojNovcica(v, n, S-v[i]) + 1);
  // vraćamo rezultat, pamteći ga pritom u nizu memo
  return memo[S] = br;
}

int main() {
  // učitavamo iznos i niz novčića
  int n, S;
  int v[MAX_V];
  cin >> S >> n;
  for(int i = 0; i < n; i++)
    cin >> v[i];
  // izračunavamo i ispisujemo minimum
  int br = minBrojNovcica(v, n, S);
  cout << (br == INF ? -1 : br)  << endl;
  return 0;
}
```

Приликом динамичког програмирања навише низ попуњавамо слева надесно.

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int S){
  int dp[MAX_S + 1];
  // iznos 0 se naplaćuje sa 0 novčića
  dp[0] = 0;
  // računamo minimalni broj novčića za sve ostale iznose
  for (int s = 1; s <= S; s++) {
    // minimalni broj novčića da se naplati iznos s (pretpostavljamo
    // da iznos nije moguće naplatiti)
    dp[s] = INF;
    // razmatramo sve mogućnosti za poslednji novčić
    for (int i = 0; i < n; i++)
      // proveravamo da li je iznos s moguće naplatiti novčićem i
      if (v[i] <= s)
        // ažuriramo minimum ako je to potrebno
        dp[s] = min(dp[s], dp[s-v[i]] + 1);
  }
  // vraćamo rezultat za iznos S
  return dp[S];
}

int main() {
  // učitavamo iznos i niz novčića
  int S;
  cin >> S;
  int n;
  cin >> n;
  int v[MAX_V];
  for(int i = 0; i < n; i++)
    cin >> v[i];
  // izračunavamo i ispisujemo minimum
  int br = minBrojNovcica(v, n, S);
  cout << (br == INF ? -1 : br) << endl;
  return 0;
}
```

Задатак: Најдужи подниз палиндром

Написати програм којим се за дати стрингу s одређује дужину најдужег подниза ниске s који је палиндром. Подниз не мора да садржи узастопне карактере ниске, али они морају да се јављају у истом редоследу (подниз се добија брисањем произвољног броја карактера).

Улаз: Са стандардног улаза се учитава ниска s састављена само од малих слова енглеске абецеде, чија је дужина највише 5000 карактера.

Излаз: На стандардни излаз исписати само тражену дужину најдужег палиндромског подниза.

Пример

Улаз	*Излаз*
najduzipalindrom	5

Решење

Кренимо од рекурзивног решења.

- Празна ниска има само празан подниз, па је дужина најдужег палиндромског подниза једнака нули. Ниска дужине 1 је сама свој палиндромски подниз, па је дужина њеног најдужег палиндромског подниза једнака 1.
- Ако ниска има бар два карактера, онда рамзатрамо да ли су њен први и последњи карактер једнаки. Ако јесу, онда они могу бити део најдужег палиндромског подниза и проблем се своди на проналажење најдужег палиндромског подниза дела ниске без првог и последњег карактера. У супротном они не могу истовремено бити део најдужег палиндромског подниза и потребно је елиминисати бар један од њих. Проблем, дакле, сводимо на проналажење најдужег палиндромског подниза суфикса ниске без првог карактера и на проналажење најдужег палиндромског подниза префикса ниске без последњег карактера. Дужи од та два палиндромска подниза је тражени палиндромски подниз целе ниске.

Овим је практично дефинисана рекурзивна процедура којом се решава проблем. У сваком рекурзивном позиву врши се анализа неког сегмента (низа узастопних карактера полазне ниске), па је сваки рекурзивни позив одређен са два броја који представљају границе тог сегмента. Ако са $f(l, d)$ означимо дужину најдужег палиндромског подниза дела ниске $s[l, d]$, тада важе следеће рекурентне везе.

$$
\begin{aligned}
f(l,d) &= 0, \quad \text{за } l > d \\
f(l,d) &= 1, \quad \text{за } l = d \\
f(l,d) &= 2 + f(l+1, d-1), \quad \text{за } l < d \text{ и } s_l = s_d \\
f(l,d) &= \max\left(f(l+1,d), f(l,d-1)\right), \quad \text{за } l < d \text{ и } s_l \neq s_d
\end{aligned}
$$

На основу овога, функцију је веома једноставно имплементирати.

```
#include <iostream>
#include <string>

using namespace std;

int najduziPalindrom(const string& s, int l, int d) {
  if (l > d)
    return 0;
  if (l == d)
    return 1;
  if (s[l] == s[d])
    return 2 + najduziPalindrom(s, l+1, d-1);
  return max(najduziPalindrom(s, l, d-1),
             najduziPalindrom(s, l+1, d));
}

int najduziPalindrom(const string& s) {
  return najduziPalindrom(s, 0, s.length() - 1);
}

int main() {
   string s;
   cin >> s;
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

У претходној функцији долази до преклапања рекурзивних позива, па је пожељно употребити мемоизацију. За мемоизацију користимо матрицу (практично, њен горњи троугао у којем је $l < d$).

```
#include <iostream>
#include <string>
#include <vector>
```

```
#include <algorithm>

using namespace std;

int najduziPalindrom(const string& s, int l, int d,
                     vector<vector<int>>& memo) {
  if (memo[l][d] != -1)
    return memo[l][d];
  if (l > d)
    return memo[l][d] = 0;
  if (l == d)
    return memo[l][d] = 1;
  if (s[l] == s[d])
    return memo[l][d] = 2 + najduziPalindrom(s, l+1, d-1, memo);
  return memo[l][d] = max(najduziPalindrom(s, l, d-1, memo),
                          najduziPalindrom(s, l+1, d, memo));
}

int najduziPalindrom(const string& s) {
  vector<vector<int>> memo(s.length(), vector<int>(s.length(), -1));
  return najduziPalindrom(s, 0, s.length() - 1, memo);
}

int main() {
   string s;
   cin >> s;
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

До ефикасног решења можемо доћи и динамичким програмирањем одоздо навише. Елемент на позицији (l, d) матрице зависи од елемената на позицијама $(l+1, d)$, $(l, d-1)$ и $(l+1, d-1)$, док се коначно решење налази у горњем левом углу матрице, тј. на пољу $(0, n-1)$. Због оваквих зависности матрицу не можемо попуњавати ни врсту по врсту, ни колону по колону, већ дијагоналу по дијагоналу. На дијагоналу испод главне уписујемо све нуле, на главну дијагоналу све јединице, а затим попуњавамо једну по једну дијагоналу изнад главне, све док не дођемо до елемента у горњем левом углу.

Прикажимо како изгледа попуњена матрица на примеру ниске `abaccba`.

```
    abaccba
    0123456
    -------
a 0|1133346
b 1|0111244
a 2| 011224
c 3|  01222
c 4|   0111
b 5|    011
a 6|     01
```

Коначно решење 6 одговара поднизу `abccba`.

Ово решење има и меморијску и временску сложеност $O(n^2)$.

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;
```

```
int najduziPalindrom(const string& s) {
  int n = s.length();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i].resize(n, 0);
    dp[i][i] = 1;
  }
  for (int k = 1; k < n; k++)
    for (int l = 0; l + k < n; l++) {
      int d = l + k;
      if (s[l] == s[d])
        dp[l][d] = dp[l+1][d-1] + 2;
      else
        dp[l][d] = max(dp[l+1][d], dp[l][d-1]);
    }

  return dp[0][n - 1];
}

int main() {
   string s;
   cin >> s;
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

Меморијску сложеност је могуће редуковати. Примећујемо да елементи сваке дијагонале зависе само од елемената претходне две дијагонале. Могуће је да чувамо само две дијагонале - текућу и претходну. Током ажурирања текуће дијагонале њене постојеће елементе истовремено преписујемо у претходну. Када су карактери једнаки, тада у привремену промењљиву бележимо одговарајући елемент претходне дијагонале, на његово место уписујемо одговарајући елемент текуће дијагонале, а онда на место тог елемента уписујемо вредност привремене промењљиве увећану за два. Када су карактери различити одговарајући елемент текуће дијагонале уписујемо на одговарајуће место у претходној дијагонали, а на његово место уписујемо максимум те и наредне вредности текуће дијагонале.

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

int najduziPalindrom(const string& s) {
  int n = s.length();
  // elementi dve prethodne dijagonale
  vector<int> dpp(n, 0);
  vector<int> dp(n, 1);
  for (int k = 1; k < n; k++) {
    for (int l = 0; l + k < n; l++) {
      int d = l + k;
      if (s[l] == s[d]) {
        int tmp = dp[l];
        dp[l] = dpp[l+1] + 2;
        dpp[l] = tmp;
      }
      else {
        dpp[l] = dp[l];
        dp[l] = max(dp[l], dp[l+1]);
      }
```

```
    }
    dpp[n-k] = dp[n-k];
  }

  return dp[0];
}

int main() {
   string s;
   cin >> s;
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

Рецимо још и да је решење овог задатка могуће добити и свођењем на проблем одређивања најдужег заједничког подниза две ниске који је описан у задатку Најдужи заједнички подниз две ниске. Наиме, најдужи палиндромски подниз једнак је најдужем заједничком поднизу оригиналне ниске и ниске која се добија њеним обртањем. Сложеност ове редукције је иста као и сложеност директног решења (временски $O(n^2)$, а просторно $O(n)$).

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1]) {
        dp[j] = prethodni + 1;
      } else {
        dp[j] = max(dp[j-1], dp[j]);
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int najduziPalindrom(const string& s) {
  string sObratno = s;
  reverse(begin(sObratno), end(sObratno));
  return najduziZajednickiPodniz(s, sObratno);
}

int main() {
   string s;
   cin >> s;
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

Задатак: Максимални збир сегмента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Максимални суфикси

С обзиром на то да се суфикси првих i елемената низа могу анализирати инкрементално (суфикси $i+1$ елемената низа се добијају проширивањем суфикса i елемената низа додатним елементом), проблем анализе свих сегмената је пожељно свести на проблем анализе суфикса. У овом конкретном проблему, можемо приметити да је максимални збир сегмента уједно и максимални збир суфикса који се завршава на позицији на којој се тај сегмент завршава. Стога се задатак може свести на то да се за сваку позицију у низу одреди максимални суфикс, а да се онда међу максималним суфиксима за сваку позицију пронађе онај који је глобално максимални.

Једини суфикс првих нула елемената низа је празан и његов збир је по дефиницији 0. Сви суфикси првих $i+1$ елемената низа, изузев празног суфикса добијају се додавањем елемента на позицији i на крај неког суфикса првих i елемената низа. Међу непразним суфиксима највећи збир има онај који је добијен додавањем последњег елемента управо на суфикс првих i елемената низа који има максимални збир. Од њега једино може бити већи збир празног суфикса (и то када се након проширивања последњим елементом добије негативни збир). Ако вредности максималног збира суфикса памтимо у низу, тада низ лако попуњавамо на основу веза $S_0 = 0$ и $S_{i+1} = \max(S_i + a_i, 0)$, где је са S_i обележена вредност максималног збира суфикса првих i елемената низа a.

На крају налазимо максимум низа S_i.

Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности S_i.

```
i    a_{i+1}   S_i
0              0
1    -2        0 = max(0+(-2), 0)
2     3        3 = max(0+3, 0)
3     2        5 = max(3+2, 0)
4    -3        2 = max(5+(-3), 0)
5    -3        0 = max(2+(-3), 0)
6    -2        0 = max(0+(-2), 0)
7     4        4 = max(0+4, 0)
8     5        9 = max(4+5, 0)
9    -8        1 = max(9+(-8), 0)
10    3        4 = max(1+3, 0)
```

Максимална вредност у колони S_i је 9.

Пошто користимо низ максималних збирова суфикса, меморијска сложеност је $O(n)$. Низ попуњавамо елемент по елемент, инкрементално, у једном пролазу за шта је довољно n корака, а затим максимум налазимо у новом пролазу тј. у нових n корака. Укупна сложеност је, дакле, линеарна тј. $O(n)$.

```
// maksimalni sufiks prvih i elemenata niza
vector<int> maxSufiks(n+1);
maxSufiks[0] = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  maxSufiks[i+1] = max(maxSufiks[i] + x, 0);
}

// maksimalni segment je maksimalni od svih maksimalnih sufiksa
cout << *max_element(begin(maxSufiks), end(maxSufiks)) << endl;
```

Каданов алгоритам

Максималне вредности збирова суфикса не морамо да памтимо у низу, ако њихов максимум одређујемо истовремено са одређивањем вредности максималних збирова суфикса. Овај алгоритам познат је под именом *Каданов алгоритам*.

Један начин да се дође до тог алгоритма је следећи. Покушавамо да алгоритам заснујемо на индуктивној конструкцији.

- За празан низ, једини сегмент је празан и његов је збир нула (то је уједно највећи збир који се може добити).
- Сматрамо да умемо да проблем решимо за произвољан низ дужине n и на основу тога покушавамо да решимо задатак за низ дужине $n + 1$ (полазни низ проширен једним додатним елементом).

  Сегмент највећег збира у проширеном низу се или цео садржи у полазном низу дужине n или чини суфикс проширеног низа, тј. завршава се на последњој позицији (укључујући и могућност да је ту и празан сегмент).

  На основу индуктивне хипотезе знамо да израчунамо највећи збир сегмента низа дужине n и потребно је да још одредимо максимални збир суфикса проширеног низа. Један начин да се то уради је да приликом сваког проширења низа изнова анализирамо све сегменте који се завршавају на текућој позицији, али чак иако то радимо инкрементално (кренувши од празног суфикса, па додајући уназад један по један елемент) највише што можемо добити је алгоритам квадратне сложености (пробајте да се уверите да је то заиста тако). Кључни увид је то да највећи збир суфикса који се завршава на текућој позицији можемо инкрементално израчунати знајући највећи збир суфикса низа пре проширења. Највећи збир неког непразног суфикса који се завршава на текућој позицији је збир текућег елемента низа и највећег збира неког суфикса који се завршава на претходној позицији. Од њега може бити повољнији само празан суфикс (и то само ако је претходни збир негативан).

Дакле, ако са S_i обележимо максимални збир неког суфикса првих i елемената низа, а са M_i максимални збир неког сегмента прих i елемената низа, важи да је $M_0 = S_0 = 0$, да је $S_{i+1} = \max(S_i + a_i, 0)$ и $M_{i+1} = \max(M_i, S_{i+1})$.

Имплементацију можемо направити итеративним алгоритмом коме је инваријанта да у сваком кораку петље знамо ове две вредности (максимум сегмента и максимум суфикса).

Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности S_i и M_i.

```
i    a_{i+1}   S_i                    M_i
0              0                      0
1    -2        0 = max(0+(-2), 0)     0 = max(0, 0)
2     3        3 = max(0+3, 0)        3 = max(0, 3)
3     2        5 = max(3+2, 0)        5 = max(3, 5)
4    -3        2 = max(5+(-3), 0)     5 = max(5, 2)
5    -3        0 = max(2+(-3), 0)     5 = max(5, 0)
6    -2        0 = max(0+(-2), 0)     5 = max(5, 0)
7     4        4 = max(0+4, 0)        5 = max(5, 4)
8     5        9 = max(4+5, 0)        9 = max(5, 9)
9    -8        1 = max(9+(-8), 0)     9 = max(9, 1)
10    3        4 = max(1+3, 0)        9 = max(9, 4)
```

Пошто елементе учитавамо један по један и не памтимо их истовремено, меморијска сложеност је $O(1)$. Максимални збир сегмента и суфикса инкрементално израчунавамо једним проласком кроз задате елементе и временска сложеност је линеарна тј. $O(n)$.

Приметимо да смо у претходном разматрању проширили индуктивну хипотезу претпостављајући да поред тражене вредности тј. максимума неког сегмента првих n елемената низа знамо додатно и вредност максималног суфикса првих n елемената низа.

```
int maxSufiks = 0, max = maxSufiks;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  maxSufiks += x;
```

```
  if (maxSufiks < 0)
    maxSufiks = 0;
  if (maxSufiks > max)
    max = maxSufiks;
}
cout << max << endl;
```

Задатак: Оптимално множење матрица

Множење матрица димензије $D_1 \times D_2$ и димензије $D_2 \times D_3$ даје матрицу димензије $D_1 \times D_3$ и да би се оно спровело потребно је $D_1 \cdot D_2 \cdot D_3$ множења бројева. Када је потребно измножити дужи низ матрица, онда ефикасност зависи од начина како се те матрице групишу (множење је асоцијативна операција и допуштено је било које груписање множења). Напиши програм који одређује најмањи број множења бројева потребан да би се измножио низ матрица датих димензија.

Улаз: Са стандардног улаза се прво учитава број $3 \leq n \leq 100$, а затим и низ димензија $D_0, D_1, D_2, \ldots D_{n-1}$. Матрице које се множе су димензија $D_0 \times D_1$, $D_1 \times D_2$, $\ldots D_{n-2} \times D_{n-1}$.

Излаз: На стандардни излаз исписати тражени најмањи број множења.

Пример

Улаз	*Излаз*
4 5 4 6 2	88

Објашњење: Множимо матрицу A димензије 5×4, матрицу B димензије 4×6 и матрицу C димензије 6×2. За рачунање производа $(AB)C$ потребно је $5 \cdot 4 \cdot 6 + 5 \cdot 6 \cdot 2 = 180$ операција, а за рачунање производа $A(BC)$ потребно је $4 \cdot 6 \cdot 2 + 5 \cdot 4 \cdot 2 = 88$ операција множења бројева.

Задатак: Максимални збир на путу кроз матрицу

У табели димензија $n \times n$ поља су попуњена цифрама од 0 до 9. Играч који се налази у горњем левом углу табеле може да у једном кораку пређе у суседно десно поље или суседно доње поље. Циљ му је да стигне до доњег десног поља тако да збир вредности на пређеним пољима буде максималан. Написати програм којим се одређује максимални збир коју може остварити играч при кретању од горњег левог до доњег десног угла.

Улаз: У првој линији стандардног улаза се уноси број редова табеле n ($1 \leq n \leq 30$), а у следећих n редова по n цифара од 0 до 9.

Излаз: У првој линији стандардног излаза приказати тражену вредност максималног збира.

Пример

Улаз	*Излаз*
5 4 3 5 7 5 1 9 4 1 3 2 3 5 1 2 1 3 1 2 0 4 6 7 2 1	38

Задатак: Максимални пут кроз матрицу

У табели димензија $n \times n$ поља су попуњена цифрама од 0 до 9. Играч који се налази у горњем левом углу табеле може да у једном кораку пређе у суседно десно поље или суседно доње поље. Циљ му је да стигне до доњег десног поља тако да збир вредности на пређеним пољима буде максималан. Написати програм којим се одређује максимални збир коју може остварити играч при кретању од горњег левог до доњег десног угла и инструкције кретања: `desno`, `dole`, којима се обезбеђује кретање преко поља која дају максимални збир (ако има више могућих путева, исписати било који).

Улаз: У првој линији стандардног улаза се уноси број редова табеле n ($1 \leq n \leq 30$), а у следећих n редова по n цифара од 0 до 9.

Излаз: У првој линији стандардног излаза приказати тражену вредност максималног збира, а у наредним линијама исписати инструкције за кретање: `desno`, `dole` - у сваком реду по једну, којима се обезбеђује кретање преко поља која дају максимални збир.

Пример

Улаз	*Излаз*
5 4 3 5 7 5 1 9 4 1 3 2 3 5 1 2 1 3 1 2 0 4 6 7 2 1	38 desno dole dole dole dole desno desno desno

Решење

Овај задатак представља проширење задатка Максимални збир на путу кроз матрицу. Након одређивања оптималног пута, потребно је реконструисати и сам пут. Иако је понекад потребно складиштити и додатне информације да би се од вредности решења могло реконструисати решење, овде то није случај - решење је у потпуности могуће реконструисати на основу матрице изграђене у склопу динамичког програмирања навише (или у склопу мемоизације). Ипак, нагласимо да нам је потребна цела матрица и да није могуће извршити оптимизацију у којој се чува само текућа врста матрице.

Током реконструкције решења крећемо уназад, од завршног поља па све до почетног и на основу вредности у матрици закључујемо са ког претходног поља се стигло на текуће. Анализирамо поље лево и поље изнад текућег и гледамо на ком од њих пише већа вредност (ако су вредности једнаке, свеједно је које ћемо поље одабрати). Када знамо са ког смо поља стигли на текуће, знамо и на основу које инструкције робот прави тај корак (ако смо дошли са поља лево, инструкција је `desno`, а ако смо дошли са поља изнад, инструкција је `dole`). Потребно је само да посебно обратимо пажњу на поља у првој врсти и првој колони, јер код њих постоји само једна могућност. Крај реконструкције наступа када дођемо до поља $(0, 0)$. Приметимо да на овај начин, крећући се од завршног ка полазном пољу одређујемо низ инструкција унатраг, у обратном редоследу. Да бисмо добили инструкције у редоследу који је захтеван, можемо употребити рекурзивну функцију, тако што прво рекурзивно испишемо све инструкције за долазак на претходно поље и тек након тога испишемо инструкцију за прелазак са претходног на текуће поље.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// ucitavanje matrice sa standardnog ulaza
vector<vector<int>> ucitaj() {
  int n;
  cin >> n;
  vector<vector<int>> M(n);
  for (int i = 0; i < n; i++) {
    M[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> M[i][j];
  }
  return M;
}

// odredjujemo matricu koja na polju (v, k) sadrzi duzinu
// najkraceg puta od (0, 0) do (v, k)
vector<vector<int>> maksZbirDP(const vector<vector<int>>& M) {
  int n = M.size();
  vector<vector<int>> dp(n);
```

```
  for (int i = 0; i < n; i++)
    dp[i].resize(n);
  dp[0][0] = M[0][0];
  for (int v = 1; v < n; v++)
    dp[v][0] = dp[v-1][0] + M[v][0];
  for (int k = 1; k < n; k++)
    dp[0][k] = dp[0][k-1] + M[0][k];
  for (int v = 1; v < n; v++)
    for (int k = 1; k < n; k++)
      dp[v][k] = max(dp[v-1][k] + M[v][k], dp[v][k-1] + M[v][k]);
  return dp;
}

// ispisuje instrukcije kretanja od polja (0, 0) do polja (v, k)
void pisiPut(int v, int k, const vector<vector<int>>& dp) {
  // ako smo vec na polju (0, 0), nema potrebe da se pomeramo
  if (v == 0 && k == 0)
    return;

  if (v == 0) {
    // na polja u prvoj vrsti mozemo stici samo sa polja levo od njih,
    // pomerajuci se desno
    pisiPut(v, k-1, dp);
    cout << "desno" << endl;
  } else if (k == 0) {
    // na polja u prvoj koloni mozemo stici samo sa polja iznad njih,
    // pomerajuci se na dole
    pisiPut(v-1, k, dp);
    cout << "dole" << endl;
  } else
    // u suprotnom analiziramo da li nam je povoljnije da stignemo sa
    // polja iznad ili sa polja levo
    if (dp[v-1][k] > dp[v][k-1]) {
      pisiPut(v-1, k, dp);
      cout << "dole" << endl;
    } else {
      pisiPut(v, k-1, dp);
      cout << "desno" << endl;
    }
}

int main() {
  vector<vector<int>> M = ucitaj();
  int n = M.size();
  vector<vector<int>> dp = maksZbirDP(M);
  cout << dp[n-1][n-1] << endl;
  pisiPut(n-1, n-1, dp);
  return 0;
}
```

Уместо рекурзије можемо употребити и стек. Током пута уназад инструкције за прелазак са претходног на текуће поље ћемо постављати на помоћни стек, а затим, када стигнемо до поља $(0, 0)$, скидаћемо једну по једну вредност са стека и исписиваћемо је.

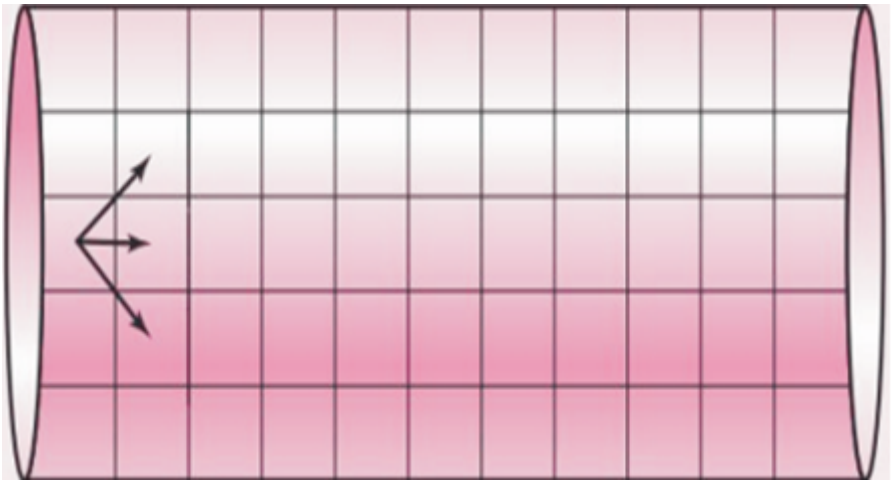
```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
```

```
using namespace std;

// ucitavanje matrice sa standardnog ulaza
vector<vector<int>> ucitaj() {
  int n;
  cin >> n;
  vector<vector<int>> M(n);
  for (int i = 0; i < n; i++) {
    M[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> M[i][j];
  }
  return M;
}

// odredjujemo matricu koja na polju (v, k) sadrzi duzinu
// najkraceg puta od (0, 0) do (v, k)
vector<vector<int>> maksZbirDP(const vector<vector<int>>& M) {
  int n = M.size();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n);
  dp[0][0] = M[0][0];
  for (int v = 1; v < n; v++)
    dp[v][0] = dp[v-1][0] + M[v][0];
  for (int k = 1; k < n; k++)
    dp[0][k] = dp[0][k-1] + M[0][k];
  for (int v = 1; v < n; v++)
    for (int k = 1; k < n; k++)
      dp[v][k] = max(dp[v-1][k] + M[v][k], dp[v][k-1] + M[v][k]);
  return dp;
}

// ispisuje instrukcije kretanja
void pisiPut(const vector<vector<int>>& dp, int n) {
  // instrukcije dobijamo u obratnom redosledu, pa ih obrcemo
  // koriscenjem pomocnog steka
  stack<string> put;

  // krecemo od kraja i idemo ka pocetku
  int v = n-1, k = n-1;
  while (v > 0 || k > 0) {
    if (v == 0) {
      // na polja u prvoj vrsti mozemo stici samo sa polja levo od njih,
      // pomerajuci se desno
      k--;
      put.push(”desno”);
    } else if (k == 0) {
      // na polja u prvoj koloni mozemo stici samo sa polja iznad njih,
      // pomerajuci se na dole
      v--;
      put.push(”dole”);
    } else
      // u suprotnom analiziramo da li nam je povoljnije da stignemo sa
      // polja iznad ili sa polja levo
      if (dp[v-1][k] > dp[v][k-1]) {
        v--;
        put.push(”dole”);
```

```
      } else {
        k--;
        put.push("desno");
      }
  }

  // sadrzaj steka ispisujemo u obratnom redosledu
  while (!put.empty()) {
    cout << put.top() << endl;
    put.pop();
  }
}

int main() {
  vector<vector<int>> M = ucitaj();
  int n = M.size();
  vector<vector<int>> dp = maksZbirDP(M);
  cout << dp[n-1][n-1] << endl;
  pisiPut(dp, n);
  return 0;
}
```

Задатак: Цилиндрична матрица

Елементи целобројне матрице $A_{m \times n}$ су савијени у цилиндар тако да се горња и доња (прва и последња) врста матрице додирују. Ако се робот креће са левог краја цилиндра (од прве колоне) ка десном (до последње колоне), уз дозвољена кретања једно поље горе-десно, десно, доле-десно, одредити пут којим робот може да пређе да би обезбедио најмањи збир вредности поља на том путу.



Слика 7.1: слика

Улаз: У првој линији стандардног улаза уноси се број редова табеле m ($1 \leq m \leq 30$), у другој број колона табеле n ($1 \leq n \leq 30$), а у следећих m редова по n вредности од 0 до 100.

Излаз: На стандардном излазу у првој линији приказати тражену вредност максималног збира, а у следећих m редова индекс врсте и колоне (раздвојене празнином) поља преко којих пролази робот. Ако постоји више путева минималног збира, исписати било који од њих.

Пример

Улаз	*Излаз*
5	8
6	1 0
4 3 5 7 5 8	0 1
1 9 4 1 3 9	4 2
2 3 9 1 2 5	3 3
1 7 8 2 0 1	3 4
4 6 1 9 1 7	3 5

Објашњење

Робот прелази преко следећих поља (прелаз са поља (0, 1) на поље (4, 2) могуће је захваљујући цилиндричном облику матрице):

```
. 3 . . . .
1 .
. . . . . .
. . . 2 0 1
. . 1 . . .
```

Задатак: Подскуп максималног збира дељивог са k

Дат је скуп S од n различитих природних бројева. Написати програм који проналази максималну суму подскупа A скупа S која је дељива са k.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 10^5$). Свака од следећих n линија садржи један елементе скупа, природан број који није већи од 200. Последња линија садржи природни број k ($2 \leq k \leq 100$).

Излаз: На стандардном излазу у једној линији приказати тражену максималну суму подскупа која је дељива са k, ако такав подскуп не постоји приказати 0.

Пример

Улаз	*Излаз*
4	18
10	
2	
7	
6	
3	

Решење

Решење грубом силом подразумева да се испитају сви подскупови. Набрајање сви подскупова може се вршити рекурзивном функцијом, као у задатку Сви подскупови. Обрада сваког подскупа ће се састојати од тога да се провери да ли му је збир елемената дељив са k и да ли је већи од дотада пронађеног максимума (који одржавамо у променљивој коју прослеђујемо функцији и коју ажурирамо сваки пут када наиђемо на подскуп већег збира дељивог са k). Да бисмо избегли израчунавање збира елемената за сваки пронађени подскуп, рекурзивној функцији уместо низа елемената подскупа прослеђујемо само њихов збир.

Иако би се овај алгоритам могао додатно оптимизовати одсецањем у претрази, постоје ефикаснија решења.

```
#include <iostream>
#include <vector>

using namespace std;

// funkcija izracunava maksimalnu sumu elemenata niza a koja je
// deljiva sa 3, pri cemu je n duzina prefiksa niza koji jos nije
// analiziran, tekuca suma je suma sufiksa niza nakon tih pocetnih n
// elemenata, a maxSuma je maksimalna suma elemenata niza a koja je
// deljiva sa 3 medju svim kompletnim podskupovima koji su do sada
// ispitani
void maxSumaDeljivaSaK(const vector<int>& a, int n, int k,
```

```
                      int tekucaSuma, int &maxSuma) {
  // ako nije preostalo vise elemenata
  if (n == 0) {
    // podskup je kompletiran, suma svih njegovih elemenata jednaka je
    // tekucoj sumi, pa azuriramo maxSumu ako je to potrebno
    if (tekucaSuma % k == 0 && tekucaSuma > maxSuma)
      maxSuma = tekucaSuma;
    return;
  }
  // poslednji element niza a (element a[n-1]) nije ukljucen u podskup
  maxSumaDeljivaSaK(a, n-1, k, tekucaSuma, maxSuma);
  // poslednji element niza a (element a[n-1]) jeste ukljucen u podskup
  maxSumaDeljivaSaK(a, n-1, k, tekucaSuma + a[n-1], maxSuma);
}

// funkcija izracunava maksimalnu sumu elemenata niza a koja je
// deljiva sa 3 u nizu a duzine n
int maxSumaDeljivaSaK(const vector<int>& a, int n, int k) {
   int maxSuma = 0;
   maxSumaDeljivaSaK(a, n, k, 0, maxSuma);
   return maxSuma;
}

int main() {
  // ucitavamo elemente niza
   int n;
   cin >> n;
   vector<int> a(n);
   for (int i = 0; i < n; i++)
     cin >> a[i];
   int k;
   cin >> k;
   // odredjujemo i ispisujemo trazenu maksimalnu sumu
   cout << maxSumaDeljivaSaK(a, n, k) << endl;
   return 0;
}
```

Задатак можемо решити рекурзивном претрагом, тако што уопштимо проблем који решавамо и дефинишемо функцију која проналази највећи збир подскупа неког префикса елемената полазног низа, такав да тај збир при дељењу са k даје неки дати остатак (задати број o између 0 и $k-1$).

Нагласимо да када је o различито од нуле, такав подскуп не мора да постоји (у случају када је остатак 0, то се не може десити, јер је збир празног подскупа једнак 0). Функцију можемо дефинисати тако да врати 0 и у случају када тражени подскуп не постоји и тада лако можемо детектовати да ли је тражени подскуп пронађен или он не постоји (просто, тако што израчунамо остатак повратне вредности функције при дељењу са k и упоредимо са траженим остатком - ако је функција вратила 0, а остатак је различит од 0, тада је јасно да функција није вратила одговарајући збир подскупа тј. подскуп не постоји). Зато се излаз из рекурзије дефинише тако да када је префикс празан, функција увек враћа вредност 0.

Ако префикс није празан, анализира се случај када се последњи елемент префикса (нека је то елемент a_{n-1}) укључује у подскуп и када се не укључује у подскуп.

Ако последњи елемент није укључен, онда међу елементима префикса из ког је избачен тај последњи елемент првим рекурзивним позивом тражимо највећи збир подскупа који при дељењу са k даје остатак o (ако такав подскуп не постоји, добићемо вредност 0).

Ако у подскуп укључујемо a_{n-1}, да би збир елемената подскупа при дељењу са k дао остатак o, збир елемената подскупа без елемента a_{n-1} треба да да остатак који се може израчунати као разлика по модулу k броја o и остатка при дељењу са k броја a_{n-1}. На пример, ако нам треба остатак 1 при дељењу са 3, а број a_{n-1} даје остатак 2 при дељењу са 3, тада остали елементи подскупа треба заједно да такође дају остатак 2 при дељењу са 3 (јер је $(1-2) \bmod 3 = 2$, тј $(2+2) \bmod 3 = 1$). У задатку Монопол приказано је да се разлика

бројева a и b по модулу k може израчунати као $(a \bmod k - b \bmod k + k) \bmod k$ (додавањем k избегава се рад са негативним бројевима). Пошто је $o \bmod k = k$, тада можемо извршити други рекурзивни позив у ком ће се унутар елемената префикса скраћеног за елемент a_{n-1} тражити максимални збир чији је остатак при дељењу са k једнак $(o - a_{n-1} \bmod k + k) \bmod k$.

Ако такав подскуп постоји, тада се додавањем елемента a_{n-1} добија максимални подскуп полазног префикса чији је остатак при дељењу са k једнак o, међутим, могуће је да подскуп скраћеног префикса не постоји и тада није могуће наћи подскуп полазног префикса који укључивао a_{n-1} и давао остатак o. Уместо анализе резултата другог рекурзивног позива, можемо тај резултат увећати за a_{n-1} и остатак при дељењу са k упоредити са o. Ако је остатак једнак o и ако је тај збир већи од резултата првог рекурзивног позива (случаја без укључивања елемента a_{n-1}), онда је тако добијен збир максималан збир полазног префикса који даје остатак o и њега враћамо као резултат функције. У супротном подскуп није могуће добити укључивањем a_{n-1} или је тако добијен максимални подскуп неоптималан, па треба вратити резултат првог рекурзивног позива (нагласимо и да тај рекурзивни позив може да врати 0, што је у реду, јер тада знамо да су оба рекурзивна позива вратила 0, па подскуп чији би остатак био o није могуће направити и главни позив враћа 0, што је у складу са спецификацијом наше функције).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// vraca maksimalni zbir nekog podskupa prvih n elemenata niza a koji
// pri deljenju sa k daje ostatak ost ili 0 ako takav neprazan podskup
// ne postoji
int maksZbirPodskupaOstK(const vector<int>& a, int n, int k, int ost) {
  // razmatramo prazan skup elemenata, pa nemamo neprazan podskup
  if (n == 0)
    return 0;
  // element a[n-1] nije ukljucen u podskup, pa rezultat prenosimo iz slucaja
  // u kome razmatramo prvih n-1 elemenata skupa a
  int r = maksZbirPodskupaOstK(a, n-1, k, ost);
  // ukljucujemo element a[n-1] u podskup
  // da bi zbir ukljucujuci a[n-1] dao ostatak ost pri deljenju sa k,
  // zbir bez a[n-1] treba da da ostatak (ost - a[n-1]) mod k
  int m = a[n-1] + maksZbirPodskupaOstK(a, n-1, k, (ost - a[n-1] % k + k) % k);
  // ako smo uspesno pronasli neprazan podskup veceg zbira koji daje
  // ostatak ost, nego kada a[n-1] nije ukljucen, azuriramo maksimum
  if (m % k == ost && m > r)
    r = m;
  return r;
}

// vraca najveci zbir nekog podskupa niza a duzine n, koji je deljiv
// datim brojem k
int maksZbirPodskupaDeljivSaK(const vector<int>& a, int n, int k) {
  return maksZbirPodskupaOstK(a, n, k, 0);
}

int main() {
  // ucitavamo podatke
  int n;
  cin >> n;
  vector<int> a(n);
  for(int i = 0; i < n; i++)
    cin >> a[i];
  int k;
  cin >> k;
```

```
  // odredjujemo i ispisujemo najveci zbir nekog podskupa elemenata
  // niza a koji je deljiv sa k
  cout << maksZbirPodskupaDeljivSaK(a, n, k) << endl;
  return 0;
}
```

Пошто у функцији долази до преклапања рекурзивних позива, потребно је извршити њену оптимизацију техником динамичког програмирања. Када нам је рекурзивна функција на располагању, најједноставније решење је примена мемоизације (за мемоизацију користимо матрицу у којој за сваки остатак $0 \leq o < k$ и сваку дужину префикса $0 \leq d \leq n$ памтимо резултат рекурзивног позива.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// vraca maksimalni zbir nekog podskupa prvih n elemenata niza a koji
// pri deljenju sa k daje ostatak ost ili 0 ako takav neprazan podskup
// ne postoji
int maksZbirPodskupaOstK(const vector<int>& a, int n, int k, int ost,
                         vector<vector<int>>& memo) {
  // proveravamo da li smo ovu vrednost vec ranije racunali
  if (memo[ost][n] != -1)
    return memo[ost][n];

  // razmatramo prazan skup elemenata, pa nemamo neprazan podskup
  if (n == 0)
    return 0;

  // element a[n-1] nije ukljucen u podskup, pa rezultat prenosimo iz slucaja
  // u kome razmatramo prvih n-1 elemenata skupa a
  int r = maksZbirPodskupaOstK(a, n-1, k, ost, memo);
  // ukljucujemo element a[n-1] u podskup
  // da bi zbir ukljucujuci a[n-1] dao ostatak ost pri deljenju sa k,
  // zbir bez a[n-1] treba da da ostatak (ost - a[n-1]) mod k
  int m = a[n-1] + maksZbirPodskupaOstK(a, n-1, k,
                                        (ost + k - a[n-1] % k) % k, memo);
  // ako smo uspesno pronasli neprazan podskup veceg zbira koji daje
  // ostatak ost, nego kada a[n-1] nije ukljucen, azuriramo maksimum
  if (m % k == ost && m > r)
    r = m;
  return memo[ost][n] = r;
}

// vraca najveci zbir nekog podskupa niza a duzine n, koji je deljiv
// datim brojem k
int maksZbirPodskupaOstK(const vector<int>& a, int n, int k) {
  // matrica u kojoj pamtimo rezultate rekurzivnih poziva za razne
  // vrednosti ostataka i broja elemenata
  // -1 oznacava da vrednost nije ranije izracunata
  vector<vector<int>> memo(k, vector<int>(n+1, -1));
  return maksZbirPodskupaOstK(a, n, k, 0, memo);
}


int main() {
  // ucitavamo podatke
  int n;
```

```
  cin >> n;
  vector<int> a(n);
  for(int i = 0; i < n; i++)
    cin >> a[i];
  int k;
  cin >> k;
  // odredjujemo i ispisujemo najveci zbir nekog podskupa elemenata
  // niza a koji je deljiv sa k
  cout << maksZbirPodskupaOstK(a, n, k) << endl;
  return 0;
}
```

Задатак је могуће решити и динамичким програмирањем навише. Израчунавамо једну по једну врсту матрице, тј. префикс чије подскупове анализирамо проширујемо једним по једним елементом. Пошто не знамо који ће нам све остаци бити потребни, израчунаваћемо их све. Почетну врсту (која одговара празном префиксу) иницијализоваћемо на 0 (у случају када је остатак једнак 0, та нула представља збир елемената празног подскупа, који је максималан збир јер других подскупова нема, док у осталим случајевима та вредност по договору означава да тражени подскуп не постоји). Попуњавање сваке наредне врсте започињемо копирањем претходне (што одговара случају када елемент a_{n-1} није укључен у подскуп. Након тога, сваки елемент врсте увећавамо за a_{n-1}, израчунавамо остатак при дељењу са k тако добијеног збира и ако је збир већи од тренутног максималног збира придруженог том остатку, увећавамо тај максимални збир (то значи да се више исплати укључити елемент a_{n-1} у подскуп. Приметимо да у овом случају нисмо наметнули унапред колики остатак збира елемената подскупа без a_{n-1} треба да буде, него тек након додавања елемента a_{n-1} одређујемо колики је остатак максималног подсупа са укљученим елементом a_{n-1} и тек тада знамо шта је тумачење максималног збира који смо управо добили.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ucitavamo podatke
  int n;
  cin >> n;
  vector<int> a(n);
  for(int i = 0; i < n; i++)
    cin >> a[i];
  int k;
  cin >> k;

  // na mestu dp[ost, i] pamtimo najvecu vrednost zbira nekog
  // podskupa prvih i elemenata niza a koji pri deljenju sa k
  // daje ostatak ost (0 oznacava da ne postoji takav neprazan
  // podskup)
  vector<vector<int>> dp(k, vector<int>(n+1));
  // za 0 elemenata niza nije moguce napraviti neprazan podskup
  for(int o = 0; o < k; o++)
    dp[o][0]=0;

  // obradjujemo jedan po jedan element niza
  for (int i = 1; i <= n; i++) {
    // element a[i-1] iskljucujemo iz podskupa, pa se rezultat
    // prenosi od niza sa i-1 elemenata
    for (int o = 0; o < k; o++)
      dp[o][i] = dp[o][i-1];

    // element a[i-1] ukljucujemo u podskup i odredjujemo
    // vrednost podskupa za sve moguce ostatke
```

```
    for (int o = 0; o < k; o++) {
      int t = dp[o][i-1] + a[i-1];
      if (t > dp[t % k][i])
        dp[t % k][i] = t;
    }
  }

  // svi elementi niza su obradjeni, a ostatak je 0
  cout << dp[0][n] << endl;

  return 0;
}
```

На крају, можемо приметити да се матрица динамичког програмирања попуњава тако да елементи сваког наредног реда зависе само он њему непосредно претходног реда, а не од ранијих редова матрице. Зато можемо извршити и меморијску оптимизацију и уместо матрице користити само низове у којима чувамо вредности у претходном и текућој врсти матрице (пошто се елементи у текућој врсти попуњавају редоследом који није фиксиран унапред, а не слева надесно или здесна налево, није могуће да чувамо само један низ).

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  // ucitavamo podatke
  int n;
  cin >> n;
  vector<int> a(n);
  for(int i = 0; i < n; i++)
    cin >> a[i];
  int k;
  cin >> k;

  // tokom iteracije i naredne petlje, na mestu dp[ost] pamtimo
  // najvecu vrednost zbira nekog podskupa prvih i elemenata niza a
  // koji pri deljenju sa k daje ostatak ost (0 oznacava da ne postoji
  // takav neprazan podskup)

  // za 0 elemenata niza nije moguce napraviti neprazan podskup,
  // pa niz inicijalizujemo na nulu
  vector<int> dp(k, 0);

  for (int i = 1; i <= n; i++) {
    // vrednost niza dp u narednoj iteraciji
    // element a[i-1] iskljucujemo iz podskupa, pa se rezultat
    // prenosi od niza sa i-1 elemenata
    vector<int> dp_novo = dp;

    // element a[i-1] ukljucujemo u podskup i odredjujemo
    // vrednost podskupa za sve moguce ostatke
    for (int o = 0; o < k; o++) {
      int t = dp[o] + a[i-1];
      if (t > dp_novo[t % k])
        dp_novo[t % k] = t;
    }

    // azuriramo niz dp na osnovu niza dp_novo (swap je brzi od dodele)
    dp.swap(dp_novo);
```

```
  }

  // posle ukpno n+1 iteracija, obradjeni su svi elementi niza a i na
  // mestu 0 u nizu dp nalazi se maksimalni zbir ciji je ostatak 0
  cout << dp[0] << endl;


  return 0;
}
```

Задатак: Превоз предмета лифтом

Испред лифта познате носивости се налазе предмети познате масе. Напиши програм који одређује најмањи потребан број вожњи да би се сви предмети превезли.

Улаз: Са стандардног улаза се учитава број предмета n ($1 \leq n \leq 20$), затим масе предмета (цели бројеви између 1 и 50) и на крају носивост лифта (цео број између 50 и 120).

Излаз: На стандардни излаз исписати тражени број вожњи.

Пример

Улаз	*Излаз*
7 14 18 27 47 37 11 22 70	3

Решење

Решење грубом силом, за које смо сигурни да ће дати коректно решење је испитивање свих могућих пермутација предмета које одговарају свим могућим редоследима уласка у лифт. За сваки фиксирани редослед пакујемо предмете у лифт један по један док могу да стану. Сложеност овог приступа је $O(n! \cdot n)$. Одређивање свих пермутација радимо на неки од начина описаних у задатку Све пермутације.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;


int brojVoznji(const vector<int>& tezine, int nosivost) {
  int n = tezine.size();
  int minBrojVoznji = n;
  vector<int> permutacija(n);
  for (int i = 0; i < n; i++)
    permutacija[i] = i;
  do {
    int brojVoznji = 1;
    int uLiftu = 0;
    for (int i = 0; i < tezine.size(); i++) {
      if (uLiftu + tezine[permutacija[i]] > nosivost) {
        uLiftu = 0;
        brojVoznji++;
      }
      uLiftu += tezine[permutacija[i]];
    }
```

```
    if (brojVoznji < minBrojVoznji)
      minBrojVoznji = brojVoznji;
  } while (next_permutation(begin(permutacija), end(permutacija)));

  return minBrojVoznji;
}

int main() {
  int n;
  cin >> n;
  vector<int> tezine(n);
  for (int i = 0; i < n; i++)
    cin >> tezine[i];
  int nosivost;
  cin >> nosivost;
  cout << brojVoznji(tezine, nosivost) << endl;
  return 0;
}
```

Задатак решавамо индуктивно-рекурзивном конструкцијом. Ако се превози само један предмет тада се он превози сам, једном вожњом. Посматрамо скуп од $n > 1$ предмета. Сваки од њих може бити онај који је последњи убачен у лифт. За сваки предмет рекурзивно одређујемо најмањи број вожњи који је потребан да се превезу сви предмети без тог (имамо дакле n потпроблема димензије $n - 1$). Остаје питање да ли се тај последњи предмет може убацити у неку од већ постојећих вожњи или је потребно превести га посебно. Да бисмо једноставно могли дати одговор на ово питање, морамо ојачати индуктивну хипотезу и уз минималан број вожњи за предмете без текућег, морамо бити у могућности и да за све оптималне вожње тих предмета одредимо и ону у којој је маса предмета у последњем лифту минимална (то је уједно најмања маса у било којој вожњи, јер ако се најмања могућа маса добија у некој вожњи пре последње, увек можемо пермутовати ту вожњу са последњом). Пошто претпостављамо да је тренутни предмет тај који последњи улази у лифт, онда он или допуњује последњи лифт у коме се већ возе неки предмети испред њега, или се вози сам, у посебној вожњи. Први случај се дешава ако тај последњи од предмета стаје у последњи, најмање оптерећен лифт (ако је збир његове масе и масе предмета који су већ у том последњем лифту мањи или једнак носивости лифта), а други, ако не стаје. Пошто је последњи предмет фиксиран и обавезно се налази у последњем лифту, можемо лако одредити и најмању могућу масу у последњем лифту. У првом случају та маса је једнака збиру масе предмета који су већ у последњем лифту и масе тог предмета, док је у другом случају та маса једнака само маси тог предмета. Анализирањем свих избора последњег предмета и проналажењем минималног броја вожњи за сваки од избора и минималне масе у последњем лифту за сваки фиксирани избор последњег предмета добијамо најмањи могући број вожњи за свих n особа, као и најмању могућу масу у последњем лифту за тих n особа, чиме је начињен индуктивни корак.

Сваки подскуп можемо представити једним 32-битним неозначеним целим бројем (пошто је предмета мање од 20) тако да је бит на позицији i једнак 1 ако и само ако предмет i припада подскупу.

```
#include <iostream>
#include <vector>

using namespace std;

void brojVoznji(const vector<int>& tezine, int nosivost, unsigned podskup, int n, int& minVoznji, int& minPo
  if (n == 1) {
    // podskup sadrzi samo jedan predmet i on se sam prevozi
    minVoznji = 1;
    for (int i = 0; i < tezine.size(); i++)
      if (1 << i & podskup)
        minPoslednjaTezina = tezine[i];
  } else {
    // minimum inicijalizujemo na +beskonacno
    minVoznji = tezine.size(); minPoslednjaTezina = nosivost;
    // obradjujemo sve predmete
    for (int i = 0; i < tezine.size(); i++) {
```

```
      // ako se i-ti predmet nalazi u podskupu
      if (1 << i & podskup) {
        // odredjujemo optimum bez predmeta i
        int minVoznjiI, minPoslednjaTezinaI;
        brojVoznji(tezine, nosivost,
                   podskup & ~(1 << i), n-1, minVoznjiI, minPoslednjaTezinaI);
        if (minPoslednjaTezinaI + tezine[i] <= nosivost)
          // ako i-ti predmet staje u poslednji lift
          // dodajemo ga u poslednji lift
          minPoslednjaTezinaI += tezine[i];
        else {
          // ako i-ti predmet ne staje u poslednji lift
          // prevozimo samo njega
          minVoznjiI++;
          if (tezine[i] < minPoslednjaTezinaI)
            minPoslednjaTezinaI = tezine[i];
        }
        // azuriramo minimum ako je to potrebno
        if (minVoznjiI < minVoznji ||
            (minVoznjiI == minVoznji && minPoslednjaTezinaI < minPoslednjaTezina)) {
          minVoznji = minVoznjiI;
          minPoslednjaTezina = minPoslednjaTezinaI;
        }
      }
    }
  }
}

int brojVoznji(const vector<int>& tezine, int nosivost) {
  int n = tezine.size();
  unsigned podskup = (1 << n) - 1;
  int minVoznji, minPoslednjaTezina;
  brojVoznji(tezine, nosivost, podskup, n, minVoznji, minPoslednjaTezina);
  return minVoznji;
}

int main() {
  int n;
  cin >> n;
  vector<int> tezine(n);
  for (int i = 0; i < n; i++)
    cin >> tezine[i];
  int nosivost;
  cin >> nosivost;
  cout << brojVoznji(tezine, nosivost) << endl;
  return 0;
}
```

Пошто се у овако дефинисаној рекурзивној процедури рекурзивни позиви преклапају у имплементацији је неопходно применити технике динамичког програмирања. Једно решење је да се примени мемоизација. Имплементацију можемо мало поједноставити ако за излаз из рекурзије узмемо празан уместо једночланог скупа и кажемо да се он може превести једном вожњом у којем је маса последњег лифта једнака 0. С обзиром на то да се сви могући подскупови представљају неозначеним бројевима (и то од 0 до $2^n - 1$), за мемоизацију онда можемо употребити низ димензије 2^n. Елементи низа су парови целих бројева који могу бити имплементирани било преко библиотечких парова, било преко кориснички дефинисаних структура.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
#include <utility>

using namespace std;

// odredjuje se najmanji broj vožnji i najmanje opterećenje lifta za
// taj broj vožnji kada se voze svi predmeti iz datog podskupa
// (predstavljenog neoznačenim brojem) koji sadrži n predmeta
pair<int, int> brojVoznji(const vector<int>& tezine, int nosivost,
                          unsigned podskup, int n, vector<pair<int, int>>& memo) {
  // ako je problem već rešavan, vraćamo ranije određeno rešenje
  if (memo[podskup].first != 0)
    return memo[podskup];

  // bazni slučaj je kada je podskup prazan
  if (n == 0)
    return {1, 0};
  else {
    // minimum inicijalizujemo na +beskonačno
    pair<int, int> ret = {tezine.size(), nosivost};
    // svaki predmet može biti u poslednjem liftu
    for (int i = 0; i < tezine.size(); i++) {
      // ako predmet pripada podskupu
      if (((1 << i) & podskup) != 0) {
        // izbacujemo predmet iz podskupa i rekurzivno nalazimo
        // rešenje za podskup bez njega
        pair<int, int> minI = brojVoznji(tezine, nosivost, podskup ^ (1 << i), n-1, memo);
        if (minI.second + tezine[i] <= nosivost) {
          // ako predmet staje u poslednji lift, ubacujemo ga
          minI.second += tezine[i];
        } else {
          // ako ne staje, onda se ona sama vozi u liftu
          minI.first++;
          minI.second = tezine[i];
        }
        // ažuriramo minimum
        ret = min(ret, minI);
      }
    }

    // vraćamo rezultat, pamteći ga
    return memo[podskup] = ret;
  }
}

// odredjujemo najmanji broj vožnji za predmete date težine
int brojVoznji(const vector<int>& tezine, int nosivost) {
  int n = tezine.size();
  unsigned podskup = (1 << n) - 1;
  vector<pair<int, int>> memo(1<<n, {0, 0});
  return brojVoznji(tezine, nosivost, podskup, n, memo).first;
}

int main() {
  int n;
  cin >> n;
  vector<int> tezine(n);
  for (int i = 0; i < n; i++)
    cin >> tezine[i];
```

```
  int nosivost;
  cin >> nosivost;
  cout << brojVoznji(tezine, nosivost) << endl;
  return 0;
}
```

Задатак се може решити и динамичким програмирањем навише, при чему је тада потребно обратити пажњу на редослед обиласка подскупова тако да смо сигурни да су пре сваког скупа обрађени сви његови подскупови (један начин да то урадимо је да подскупове кодирамо неозначеним бројевима и да их обрађујемо у растућем редоследу).

Сложеност овог приступа једнака је $O(n \cdot 2^n)$ (број подскупова је 2^n и за сваки од њих се анализира сваки могући избор последњег предмета).

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;

int brojVoznji(const vector<int>& tezine, int nosivost) {
  int n = tezine.size();
  vector<pair<int, int>> dp(1 << n);
  dp[0] = {1, 0};
  for (unsigned podskup = 1; podskup < (1<<n); podskup++) {
    dp[podskup] = {n+1, 0};
    for (int i = 0; i < n; i++) {
      if (podskup & (1 << i)) {
        auto p = dp[podskup ^ (1 << i)];
        if (p.second + tezine[i] <= nosivost)
          p.second += tezine[i];
        else {
          p.first++;
          p.second = tezine[i];
        }
        dp[podskup] = min(dp[podskup], p);
      }
    }
  }
  return dp[(1<<n)-1].first;

}

int main() {
  int n;
  cin >> n;
  vector<int> tezine(n);
  for (int i = 0; i < n; i++)
    cin >> tezine[i];
  int nosivost;
  cin >> nosivost;

  cout << brojVoznji(tezine, nosivost) << endl;

  return 0;
}
```

Постоје и многе хеуристике (најчешће грамзиве) које могу дати добре процене броја вожњи, али не и потпуно тачно решење. Веома груба процена, која даје изненађујуће добра решења се добија када се укупна маса свих

предмета подели носивошћу лифта и количник се заокружи навише.

```
#include <iostream>
#include <vector>
#include <utility>
#include <numeric>

using namespace std;

int brojVoznji(vector<int>& tezine, int nosivost) {
  int ukupnaTezina = accumulate(begin(tezine), end(tezine), 0);
  int brojVoznji = (ukupnaTezina + nosivost - 1) / nosivost;
  return brojVoznji;
}

int main() {
  int n;
  cin >> n;
  vector<int> tezine(n);
  for (int i = 0; i < n; i++)
    cin >> tezine[i];
  int nosivost;
  cin >> nosivost;
  cout << brojVoznji(tezine, nosivost) << endl;
  return 0;
}
```

Једна могућа грамзива хеуристика која такође даје прилично добре резултате је да се у лифт увек ставља предмет највеће масе који може да стане.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int brojVoznji(vector<int>& tezine, int nosivost) {
  int n = tezine.size();
  sort(begin(tezine), end(tezine), greater<int>());
  vector<bool> odvezaoSe(n, false);
  int preostalo = n;
  int uLiftu = 0;
  int broj = 1;
  while (preostalo > 0) {
    int i;
    for (i = 0; i < n; i++)
      if (!odvezaoSe[i] && uLiftu + tezine[i] <= nosivost) {
        odvezaoSe[i] = true;
        preostalo--;
        uLiftu += tezine[i];
        break;
      }
    if (i == n) {
      broj++;
      uLiftu = 0;
    }
  }
  return broj;
}
```

```
int main() {
  int n;
  cin >> n;
  vector<int> tezine(n);
  for (int i = 0; i < n; i++)
    cin >> tezine[i];
  int nosivost;
  cin >> nosivost;
  cout << brojVoznji(tezine, nosivost) << endl;
  return 0;
}
```

Глава 8

Грамзиви алгоритми

Задатак: Реч у реч прецртавањем слова

Дате су две речи записане малим словима. Написати програм којим се проверава да ли се друга реч може добити прецртавањем слова (не обавезно суседних) у првој речи.

Улаз: У првој линији стандардног улаза налази се прва реч, а у другој линији друга реч.

Излаз: У првој линији стандардног излаза приказати реч `da` ако се друга реч може добити прецртавањем неких слова прве речи, иначе приказати реч `ne`.

Пример 1

Улаз	*Излаз*
`anica` `ana`	`da`

Пример 2

Улаз	*Излаз*
`anica` `cana`	`ne`

Решење

При прецртавању слова, њихов редослед се не мења што значи да и у другој речи редослед слова мора бити исти као у првој. Свако слово друге речи мора да се јави у првој. Ако постоји решење проблема, онда се решење може добити и тако што се у првој речи задржи прво појављивање првог слова друге речи (сва слова испред њега се прецртају) и остала слова друге речи потраже иза тог првог појављивања. Наиме, ако би постојало решење у којем је прво појављивање првог слова друге речи у првој речи прецртано, а задржано је неко његово касније појављивање, пошто се сва остала задржана слова друге речи у првој речи налазе иза тог каснијег појављивања првог слова, могли бисмо прецртати то задржано касније појављивање, а задржати прво појављивање и тако добити исправно прецртавање у којем је задржано баш прво појављивање. Сличан је случај и са даљим словима (увек у првој речи бирамо прво непрецртано појављивање текућег слова друге речи, јер ако реч можемо добити прецртавањем слова, можемо је добити и коришћењем те стратегије). Стога се решење заснива на једноставном грамзивом алгоритму (прецртавамо прво појављивање текућег слова друге речи у првој на које наиђемо и не разматрамо остале варијанте).

На основу овога, довољно пролазити у циклусу кроз прву реч, карактер по карактер, и проверавати да ли је текући карактер једнак карактеру који је на реду у другој речи (на почетку, на реду је први карактер са индексом 0). Ако су одговарајући карактери једнаки, прелази се на наредни карактер у обе речи, а иначе само у првој. Циклус треба прекинути када прођемо кроз све карактере бар једне речи. Ако смо прошли кроз све карактере друге речи, закључујемо да се друга реч може добити прецртавањем слова прве, иначе не.

Решење можемо реализовати помоћу два бројача и петље `while`.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  // ucitavamo dve reci
  string s1, s2;
  cin >> s1 >> s2;
```

```
  // redom prolazimo kroz slova obe reci dok ne dodjemo do kraja jedne
  // od njih
  int i = 0, j = 0;
  while (i < s1.size() && j < s2.size()) {
    // ako je tekuce slovo prve reci jednako tekucem slovu druge reci
    // onda ga zadrzavamo, a u suprotnom ga precrtavamo
    if (s1[i] == s2[j])
      // ako smo slovo zadrzali, prelazimo na naredno slovo druge reci
      j++;
    // prelazimo na naredno slovo prve reci
    i++;
  }

  // rec se moze dobiti ako i samo ako smo stigli do kraja druge reci
  // (sva slova druge reci smo pronasli u prvoj)
  if (j == s2.size())
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Можемо дефинисати и петљу `for` која пролази кроз слова прве речи.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  // ucitavamo dve reci
  string s1, s2;
  cin >> s1 >> s2;

  // tekuca pozicija u drugoj reci
  int j = 0;
  // prolazimo kroz sva slova prve reci
  for (int i = 0; i < s1.size(); i++) {
    // ako je tekuce slovo prve reci jednako tekucem slovu druge reci,
    // onda ga zadrzavamo, u suprotnom ga precrtavamo
    if (s1[i] == s2[j])
      j++;
    // ako smo stigli da kraja druge reci, druga rec se moze dobiti od
    // prve tako sto se sva slova do kraja precrtaju
    if (j == s2.size())
      break;
  }

  // rec se moze dobiti samo ako smo stigli do kraja druge reci
  if (j == s2.size())
    cout << "da" << endl;
  else
    cout << "ne" << endl;

  return 0;
}
```

Други поглед на исти овај поступак је да се за сваки карактер друге речи провери да ли постоји у првој речи и ако постоји, пронађе његово прво појављивање, при чему се претрага за првим карактером врши од почетка прве речи, а за сваким наредним од позиције иза оне на којој је претходни карактер нађен. Ако

се неки карактер не може пронаћи, тада другу реч није могуће добити од прве. Ако се сви карактери друге речи успешно пронађу, онда је другу реч могуће добити од прве. Претрагу карактера можемо вршити или алгоритмом линеарне претраге (слично као, на пример, у задатку Негативан број).

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  // ucitavamo dve reci
  string s1, s2;
  cin >> s1 >> s2;

  // tekuca pozicija u prvoj reci
  int i = 0, j;
  // trazimo jedno po jedno slovo druge reci u prvoj
  for (j = 0; j < s2.size(); j++) {
    // pretragu pokrecemo od pozicije i
    while (i < s1.size()) {
      // ako smo nasli slovo, prekidamo pretragu
      if (s2[j] == s1[i])
        break;
      i++;
    }
    // ako smo dosli do kraja prve reci, nismo nasli slovo i prekidamo
    // pretragu
    if (i == s1.size())
      break;
  }

  // rec se moze dobiti samo ako smo stigli do kraja druge reci
  if (j == s2.size())
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Линеарну претрагу у делу ниске можемо вршити и библиотечким функцијама. У језику C++ може се употребити метода `find` која враћа специјалну вредност `string::npos` ако карактер није нађен. Постоји варијанта ове методе која при карактер који се тражи и позицију од које креће претрага.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  // ucitavamo dve reci
  string s1, s2;
  cin >> s1 >> s2;

  // trazimo jedno po jedno slovo druge rece u prvoj, krenuvsi od pozicije i
  int i, j;
  for (i = 0, j = 0; j < s2.size(); i++, j++) {
    // trazimo tekuce slovo drugoj reci u prvoj, krenuvsi od pozicije i
    i = s1.find(s2[j], i);
    // ako ga nismo nasli, tada prekidamo pretragu
    if (i == string::npos)
      break;
    // prelazimo na sledecu poziciju u obe reci
```

```
  }

  // rec se moze dobiti samo ako smo stigli do kraja druge reci
  // (tada je svako njeno slovo pronadjeno u prvoj)
  if (j == s2.size())
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Задатак: Жаба на камењу

Камење је постављено дуж позитивног дела x-осе и за сваки камен је позната његова координата x. Жаба креће да скаче са првог камена који се налази у координатном почетку и жели да у што мање скокова дође до последњег камена. У сваком скоку она може да прескочи највише растојање r (а може да скочи и мање, ако је то потребно). Написати програм који одређује да ли жаба може стићи до последњег камена и ако може у колико најмање скокова то може учинити.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим у наредних n редова по један позитиван цео број (у питању је растуће сортиран низ бројева који представља координате камења). У последњем реду се налази позитиван цео број r.

Излаз: На стандардни излаз исписати најмањи број скокова потребан да жаба стигне до последњег камена или -1 ако то није могуће.

Пример

Улаз	*Излаз*
5 0 3 8 14 16 10	2

Задатак: Шаховске екипе

Шаховска екипа A је позвала на припреме шаховску екипу B. Свака екипа има исти број такмичара и за сваког такмичара је познат рејтинг. Екипа домаћина има могућност да одабере парове који ће играти у првом колу. Ако сваки играч домаћина побеђује госта који има мањи или једнак рејтинг, а губи од госта који има строго већи рејтинг, напиши програм који одређује који је највећи број победа које екипа домаћина (екипа A) може да оствари у првом колу.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим у наредом реду рејтинзи играча екипе домаћина (природни бројеви) раздвојени размацима, а у наредом реду рејтинзи играча екипе гостију (природни бројеви) раздвојени размацима.

Излаз: На стандардни излаз исписати само један број који представља највећи могући број победа домаћина.

Пример

Улаз	*Излаз*
4 2120 1985 2205 1842 2045 2100 1990 1980	3

Објашњење

Домаћин може да оствари највише три победе. На пример, играч 2205 може да добије играча 2100, играч 2120 може да добије играча 2045, а играч 1985 може да добије играча 1980.

Решење

Задатак можемо решити помоћу неколико различитих грамзивих алгоритама.

Једна могућност је да се парови формирају тако што се упари k најбољих домаћих играча са k најлошијих гостујућих играча. Та стратегија би била коректна, али њена имплементација није тривијална, јер није јасно колико максимално може да буде k.

Варијација коју ћемо једноставно имплементирати је следећа. Ако домаћин може да оствари бар једну победу, њу може да донесе најбољи домаћи играч. Наиме, ако би он изгубио свој меч, увек бисмо неког од играча који је добио меч могли заменити њиме (јер је он бољи од свих играча домаћина) и добити исти број победа. Поставља се питање са којим гостујућем играчем он треба да игра. Циљ нам је да након формирања тог пара преостану што лошији гостујући играчи, да би слабији играчи тима домаћина имали шансе да остваре победе. Јасно је да морамо да елиминишемо госте који су бољи од тог најбољег домаћег играча (јер њих нико од играча домаћина не може да победи), а од преосталих гостујућих играча можемо да изаберемо најбољег. Након елиминисања најбољег домаћег играча, свих гостију бољих од њега и госта са којим ће он да игра, проблем је сведен на проблем истог облика, али мање димензије. Излаз из овог рекурзивног поступка представља случај када су сви гостујући играчи бољи од најбољег међу преосталим домаћинима.

Приликом имплементације скупове домаћих и гостујућих играча можемо чувати у низовима уређеним у нерастућем редоследу рејтинга и алгоритам можемо реализовати техником два показивача (продискутоваћемо касније и остале могуће варијанте). Низ домаћина обилазимо редом, елемент по елемент, а низ гостију раздвајамо на оне које су елиминисани (оне који су до сада упарени и оне које нису упарени, али су бољи од текућег домаћег играча) и преостале. Одржавамо место почетка низа гостију који још нису обрађени и приликом тражења пара за текућег домаћег играча низ гостију обилазимо од те позиције. Сваког госта или елиминишемо, јер је бољи од текућег домаћег играча или га додељујемо текућем домаћем играчу и онда их обојицу елиминишемо. Нагласимо да се у имплементацији не морамо враћати на елиминисане госте, јер ако је неки гост бољи од текућег домаћина (најбољег међу преосталим), биће бољи и од свих наредних (преосталих) домаћина. Стога се оба показивача крећу само у једном смеру и сложеност фазе додељивања је линеарна. Укупним алгоритмом, дакле, доминира сложеност сортирања, па је укупна сложеност $O(n \log n)$.

Докажимо и формално коректност овакве грамзиве стратегије. Решење које претходни алгоритам даје задовољава услове задатка јер се сваки домаћин упарује са гостом која није бољи од њега (то се експлицитно проверава) и након упаривања се елиминише из разматрања, тако да смо сигурни да заиста постоји коректно упаривање за број победа који се враћа. Покажимо и да наша стратегија прави оптимални број победа за домаћу екипу. Доказ ће ићи техником размене, тј. тиме што ћемо се показати да се оптимално упаривање може трансформисати у оно добијено грамзивом стратегијом, одржавајући укупан број парова у којима домаћин побеђује (за играче домаћина који побеђују рећи ћемо да су добитно упарени). Посматрајмо неко оптимално упаривање. Нека је d_i најбољи домаћин који учествује у добитном упаривању. Ако он није укупно најбољи домаћин d_s, тада најбољи домаћин сигурно није добитно упарен. Можемо домаћина d_i избацити из добитног упаривања и њему придруженог госта g_i придружити укупно најбољем домаћину d_s (то је могуће јер је $d_s \geq d_i \geq g_i$). Такво упаривање је и даље оптимално (јер се број добитних парова за домаћина није променио). Нека је g_s гост која би био одабран стратегијом (најбољи гост која није бољи од d_s, тј. најбољи гост за кога важи $d_s \geq g_s$). Ако он није део тренутног добитног упаривања, онда госта g_i који тренутно игра са домаћином d_s можемо избацити и заменити њиме (то је могуће јер је $d_s \geq g_s$). Ако јесте распоређен да игра са неким d_j, онда можемо направити размену тако да d_s игра са g_s, а d_j са g_i. Докажимо да је ово и даље коректно упаривање. Важи да је $d_s \geq g_s$ и $d_s \geq g_i$. Пошто је g_s најбољи гост кога d_s може да победи, важи да је $g_s \geq g_i$. Зато је $d_j \geq g_s \geq g_i$. Са ове две евентуалне размене добијамо и даље оптималан распоред који је у складу са нашом стратегијом што се тиче првог пара. Настављајући размене по истом принципу (тј. на основу индуктивног аргумента), упаривање можемо трансформисати у оно формирано нашом стратегијом, задржавајући све време оптималност.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
   // ucitavamo ulazne podatke
   int n;
   cin >> n;
```

```
    vector<int> domaci(n);
    for (int i = 0; i < n; i++)
      cin >> domaci[i];
    vector<int> gosti(n);
    for (int i = 0; i < n; i++)
      cin >> gosti[i];

    // Ako domacini mogu da ostvare nekih k pobeda, onda tih k pobeda moze
    // da ostvari njihovih k najjacih igraca. Zato domacine obradjujemo u
    // opadajucem redosledu rejtinga i za svakog redom odredjujemo gosta kojeg
    // moze da pobedi. Za svakog domacina odredjujemo najjaceg gosta kojeg
    // moze da pobedi jer time slabijim igracima domace ekipe ostavljamo prostor
    // da pobede nekoga.

    // zelimo da i domacine i goste obilazimo u opadajucem redosledu rejtinga
    sort(begin(domaci), end(domaci), greater<int>());
    sort(begin(gosti), end(gosti), greater<int>());
    int brojPobeda = 0;
    // tekuci indeks domaceg i gostujuceg igraca
    int d = 0, g = 0;
    while (true) {
        // trazimo najjaceg gosta kojeg moze da pobedi domacin na poziciji d
        while (g < n && domaci[d] < gosti[g])
           g++;
        // ako takav gost ne postoji, ne mozemo povecati broj pobeda
        if (g >= n) break;
        // nasli smo gosta
        brojPobeda++;
        g++, d++;
    }
    cout << brojPobeda << endl;

    return 0;
}
```

Скренимо пажњу и на важност ефикасне имплементације. Посматрајмо разноврсности решења ради решење које је дуално оном претходно описаном. У тој грамзивој стратегији обрађујемо госте у неопадајућем редоследу рејтинга и сваком госту додељујемо што лошијег домаћина који може да га победи. Као што смо видели, у ефикасној имплементацији приликом преласка на сваког новог госта требало би домаћина тражити само међу онима који у ранијим корацима нису елиминисани (било тако што су упарени или тако што је установљено да не могу да победе неког од слабијих гостију). Ако претрагу домаћина сваки пут почињемо из почетка (водећи рачуна о томе да раније упарене домаћине не упарујемо поново, тако што у посебном низу региструјемо оне домаћине које смо већ упарили) добићемо алгоритам чија је сложеност најгорег случаја $O(n^2)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  // ucitavamo ulazne podatke
  int n;
  cin >> n;
  vector<int> domaci(n);
  for (int i = 0; i < n; i++)
    cin >> domaci[i];
  vector<int> gosti(n);
```

```
  for (int i = 0; i < n; i++)
    cin >> gosti[i];

  // Ako je moguce pobediti nekih k gostiju, onda tih k gostiju
  // mogu biti k najslabijih gostiju. Zato goste obradjujemo u
  // rastucem redosledu rejtinga i za svakog redom odredjujemo domacina koji
  // moze da pobedi tog gosta, a nije ranije uparen.
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  int brojPobeda = 0;

  vector<bool> zauzet(n, false);
  for (int g = 0; g < n; g++)
    for (int d = 0; d < n; d++)
      if (!zauzet[d] && domaci[d] >= gosti[g]) {
        brojPobeda++;
        zauzet[d] = true;
        break;
      }

  cout << brojPobeda << endl;

  return 0;
}
```

Још једна исправна варијација на тему наше победничке стратегије (тј. њене дуалне варијанте) обилази све домаћине и госте у неопадајућем редоследу рејтинга и ако домаћин може да победи најслабијег тренутно нераспоређеног госта, онда га упарујемо са њим, а у супротном га упарујемо са најјачим гостом. Ту стратегију можемо имплементирати тако што чувамо скуп преосталих домаћина и гостију, проналазимо најмањи тј. највећи елемент у скупу и уклањамо их. Ако се скуп имплементира преко низа (вектора, листе), добијамо веома неефикасан алгоритам, квадратне сложености (јер се и проналажење минимума и максимума и уклањање елемента са дате позиције врши у линеарној сложености).

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;


int main()
{
  int n;
  cin >> n;
  vector<int> domaci(n);
  for (int i = 0; i < n; i++)
    cin >> domaci[i];
  vector<int> gosti(n);
  for (int i = 0; i < n; i++)
    cin >> gosti[i];

  int brojPobeda = 0;

  // sve dok je duzina vektora razlicita od nule
  while (domaci.size() > 0) {
    int najmanjidom = *min_element(begin(domaci), end(domaci));
    auto it = find(begin(domaci), end(domaci), najmanjidom);
    domaci.erase(it);

    int najmanjigost = *min_element(begin(gosti), end(gosti));
```

```
    if (najmanjidom >= najmanjigost) {
      brojPobeda++;
      it = find(begin(gosti), end(gosti), najmanjigost);
      gosti.erase(it);
    } else {
      int najvecigost = *max_element(begin(gosti), end(gosti));
      it = find(begin(gosti), end(gosti), najvecigost);
      gosti.erase(it);
    }
  }
  cout << brojPobeda << endl;
}
```

Програм постаје много ефикаснији ако употребимо библиотечке колекције које нам пружају ефикаснију имплементацију скупа (која дозвољава ефикасно тражење и уклањање минималног и максималног елемента). У језику C++ можемо употребити мултискупове (јер можда постоји више играча са истим рејтингом) које на располагању имамо кроз колекцију `multiset`. Пошто је мултискуп уређен итератор `begin()` указује на најмањи елемент, а итератор `prev(end())` на највећи елемент. Уклањање елемента можемо извршити помоћу метода `erase`. Под претпоставком да се операције са мултискупом врше у логаритамској сложености у односу број елемената у мултискупу, овај алгоритам ће бити сложености $O(n \log n)$.

```
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;


int main()
{
  int n;
  cin >> n;
  multiset<int> domaci;
  for (int i = 0; i < n; i++) {
    int x; cin >> x;
    domaci.insert(x);
  }
  multiset<int> gosti;
  for (int i = 0; i < n; i++) {
    int x; cin >> x;
    gosti.insert(x);
  }

  int brojPobeda = 0;

  // sve dok ne rasporedimo sve domace igrace
  while (domaci.size() > 0) {
    // rasporedjujemo najboljeg domacina
    int najmanjidom = *domaci.begin();
    domaci.erase(domaci.begin());
    // sa najlosijim gostom ako moze da ga pobedi ili sa najboljim
    // gostom ako ne moze
    int najmanjigost = *gosti.begin();
    if (najmanjidom >= najmanjigost) {
      brojPobeda++;
      gosti.erase(gosti.begin());
    } else {
      gosti.erase(prev(gosti.end()));
    }
  }
```

```
  cout << brojPobeda << endl;
}
```

Ипак најефикасније решење добијамо ако мултискупове представимо сортираним низом, а брисање не вршимо ефективно, већ само чувамо показивач на тренутно необрађеног домаћина, док у скупу гостију необрађене госте чувамо између два показивача.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
  // ucitavamo ulazne podatke
  int n;
  cin >> n;
  vector<int> domaci(n);
  for (int i = 0; i < n; i++)
    cin >> domaci[i];
  vector<int> gosti(n);
  for (int i = 0; i < n; i++)
    cin >> gosti[i];

  // sortiramo oba tima u rastucemo redosledu rejtinga
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  int brojPobeda = 0;
  // pozicija najslabijeg i najjaceg nerasporedjenog gosta
  int gSlabi = 0, gJaki = n-1;
  // rasporedjujemo sve domace igrace
  for (int d = 0; d < n; d++) {
    // rasporedjujemo najboljeg domacina sa najlosijim gostom ako moze
    // da ga pobedi ili sa najboljim gostom ako ne moze
    if (domaci[d] >= gosti[gSlabi]) {
      brojPobeda++;
      gSlabi++;
    } else {
      gJaki--;
    }
  }

  cout << brojPobeda << endl;

  return 0;
}
```

За веома мале улазне примере, задатак би могао да се реши и грубом силом, тако што бисмо покушали сва могућа упаривања.

```
#include <iostream>
#include <vector>

using namespace std;

int maksPobeda(const vector<int>& domaci, vector<int>& gosti, int k = 0) {
  if (k == domaci.size())
    return 0;
  int maks = 0;
```

```
  for (int i = k; i < gosti.size(); i++) {
    swap(gosti[i], gosti[k]);
    int pobeda = (domaci[k] >= gosti[k] ? 1 : 0) +
                 maksPobeda(domaci, gosti, k+1);
    if (pobeda > maks)
      maks = pobeda;
    swap(gosti[i], gosti[k]);
  }
  return maks;
}

int main() {
   // ucitavamo ulazne podatke
   int n;
   cin >> n;
   vector<int> domaci(n);
   for (int i = 0; i < n; i++)
     cin >> domaci[i];
   vector<int> gosti(n);
   for (int i = 0; i < n; i++)
     cin >> gosti[i];

   cout << maksPobeda(domaci, gosti) << endl;
}
```

Примећујемо да у претходној претрази долази до преклапања рекурзивних позива, па ствар можемо покушати да поправимо мемоизацијом. Параметар мемоизације може бити скуп преосталих гостију, који можемо кодирити битовима неозначеног целог броја. Пошто је мемоизациона табела огромна ово решење се може употребити само за релативно мале димензије улаза (пар десетина такмичара).

Задатак: Ментори

Такмичари на сајту "Петља" решавају задатке и тако стичу свој рејтинг. Одлучено је да искуснији такмичари помажу млађима, тако што ће им бити ментори. Да би један такмичар могао да буде ментор другоме, његов рејтинг треба да буде бар два пута већи него рејтинг првога. Ако су познати рејтинзи свих такмичара, одредити који је највећи број парова ученик-ментор који се може формирати, при чему такмичар може истовремено бити и ученик и ментор (ментор ономе ко има бар два пута мањи рејтинг од њега, а ученик ономе који има бар два пута већи рејтинг од њега), али ни један такмичар не може да има два ученика нити два ментора.

Улаз: У једној линији стандардног улаза налази се број такмичара N, а затим се, у свакој од N наредних линија стандардног улаза, налази по један цео број који представља рејтинг такмичара.

Излаз: На стандардни излаз исписати само један цео број који представља највећи могући број парова који се могу формирати.

Пример

Улаз	*Излаз*
6 4 3 10 1 30 40	4

Задатак: Распоред активности

У једном кабинету се суботом одржава обука програмирања. Сваки наставник је написао термин у ком жели да држи наставу (познат је сат и минут почетка и сат и минут завршетка часа). Одреди како је могуће

направити распоред часова тако да што више наставника буде укључено.

Улаз: Са стандардног улаза се учитава прво број n (укупан број наставника, $1 \leq n \leq 50000$), а затим у n наредних редова по четири броја раздвојена размацима који представљају сат и минут почетка тј. завршетка часа (претпоставити да је завршетак увек иза почетка).

Излаз: На стандардни излаз исписати највећи број наставника који могу да одрже своје часове.

Пример

Улаз	*Излаз*
7 8 15 9 20 10 45 11 30 11 20 12 45 9 30 12 40 10 20 11 20 12 00 13 00 11 30 13 30	3

Објашњење

Могу се одржати, на пример, часови од 8:15 до 9:20, затим час од 10:20 до 11:20 и на крају од 11:30 до 13:30.

Решење

Сваки час можемо представити паром бројева који представљају број минута од претходне поноћи до почетка и до краја часа (већ приликом учитавања сате и минуте можемо превести само у минуте).

Исцрпна претрага

Наивно решење се заснива на испитивању свих могућих подскупова скупа тачака који су такви да се сви часови могу одржати (никоја два часа из тог скупа се не секу). Испитивање постојања пресека два часа се може урадити на исти начин као у задатку Интервали - пресек постоји ако и само ако је каснији почетак часа после ранијег краја часа. Генерисање свих подскупова вршимо рекурзивно, бектрекингом на исти начин као у задатку Сви подскупови. С обзиром на велики број подскупова које треба испитати ово решење је веома неефикасно (сложеност му је експоненцијална).

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <utility>

using namespace std;

// casove predstavljamo uredjenim parovima
typedef pair<int, int> cas;

cas napraviCas(int pocSat, int pocMin, int krajSat, int krajMin) {
  return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}

inline int pocetakCasa(const cas& c) {
  return c.first;
}

inline int krajCasa(const cas& c) {
  return c.second;
}


// provera da li postoji presek dva casa
bool postojiPresek(const cas& a, const cas& b) {
```

```
  return max(pocetakCasa(a), pocetakCasa(b)) < min(krajCasa(a), krajCasa(b));
}

// odredjuje najveci broj casova koji se moze dobiti tako sto se na k
// casova sa pocetka vektora zakzani dodaju casovi iz vektora casovi
// od pozicije i nadalje
int najviseCasova(const vector<cas>& casovi, int i, vector<cas>& zakazani, int k) {
  if (i == casovi.size()) {
    // ako nema vise kandidata u vektoru casovi, najvise mozemo
    // odrzati k casova (onih iz vektora zakazani)
    return k;
  } else {
    // racunamo najveci broj casova koji se moze dobiti ako se
    // preskoci cas na poziciji i
    int bezItog = najviseCasova(casovi, i + 1, zakazani, k);
    int rez = bezItog;
    // proveravamo da li se cas na poziciji i moze odrzati tako sto
    // proveravamo da li taj cas ima presek sa nekim od vec zakazanih
    // casova
    bool mozeIti = true;
    for (int j = 0; j < k; j++)
      if (postojiPresek(casovi[i], zakazani[j])) {
        mozeIti = false;
        break;
      }
    // ako se moze odrzati i-ti cas
    if (mozeIti) {
      // dodajemo ga na kraj vektora zakazanih casova
      zakazani[k] = casovi[i];
      // proveravamo koliko se najvise casova moze odrzati kada je on
      // ukljucen
      int saItim = najviseCasova(casovi, i + 1, zakazani, k + 1);
      // rezultat je veci od broja casova bez i-tog i sa i-tim
      rez = max(rez, saItim);
    }
    return rez;
  }
}

// najveci broj casova koji se mogu odrzati
int najviseCasova(const vector<cas>& casovi) {
  vector<cas> zakazani(casovi.size());
  return najviseCasova(casovi, 0, zakazani, 0);
}

int main() {
  int n;
  cin >> n;
  vector<cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    casovi[i] = napraviCas(pocSat, pocMin, krajSat, krajMin);
  }
  cout << najviseCasova(casovi) << endl;

  return 0;
}
```

Грамзиви приступ

Ефикасније решење се може добити грамзивим приступом. Часове можемо сортирати неопадајуће на основу времена њиховог завршетка и обилазити их у том редоследу. Од свих нераспоређених часова бирамо онај који се најраније завршава и који се може одржати (не сече се са до сада одржаним часовима, тј. почиње након завршетка претходног часа). Интуитивно, таквим избором остављамо што већу могућност за распоређивање накнадних часова.

Формално, претпоставимо да је $O = \{c_1, \ldots, \ldots, c_k\}$, скуп часова који представља неко оптимално решење, при чему су часови c_1 до c_k сортирани неопадајуће по редоследу њиховог завршетка. Пошто се сви ти часови могу одржати, између њих нема преклапања и сваки наредни почиње након завршетка претходног. Нека је c_i први час у овом скупу који не би био изабран нашом стратегијом. Претпоставимо да би наша стратегија уместо њега одабрала час c_i'. Покажимо да се заменом часа c_i часом c_i' добија такође распоред који је оптималан. Покажимо да се c_i завршава после c_i'. Заиста, ако је $i = 0$, тада наша стратегија бира c_i' који се први завршава, па се стога c_i не може завршавати пре њега. Ако је $i > 0$, тада знамо да се c_i мора да почиње после c_{i-1}, међутим, наша стратегија за c_i' бира онај час који почиње након c_{i-1} који се први завршава, па се c_i ни у овом случају не може завршавати пре c_i'. Ако постоје часови у O пре часа c_i, они остају непромењени и час c_i' се не крши са њима (јер је одабран тако да почиње након завршетка часа c_{i-1}). Пошто се c_i' не завршава касније него c_i он се сигурно не судара ни са једним часом из O који иде после c_i (јер сви они почињу и завршавају се након краја часа c_i). Дакле, када се c_i замени са c_i' и даље се добија исправан распоред са истим бројем одржаних часова као O за који смо претпоставили да је оптималан. По истом приницпу можемо мењати наредне часове (ово се мора зауставити јер у сваком наредном оптималном скупу имамо по један час више који је у складу са нашом стратегијом) и тако показати да ће наша стратегија вратити оптималан скуп.

Рецимо да постоји и дуално решење у којем се бира онај час који последњи почиње и часови се обилазе уназад, по нерастућем редоследу њиховог почетка.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <utility>

using namespace std;

// casove predstavljamo uredjenim parovima
typedef pair<int, int> cas;

cas napraviCas(int pocSat, int pocMin, int krajSat, int krajMin) {
  return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}

inline int pocetakCasa(const cas& c) {
  return c.first;
}

inline int krajCasa(const cas& c) {
  return c.second;
}

int main() {
  int n;
  cin >> n;
  vector<cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    casovi[i] = napraviCas(pocSat, pocMin, krajSat, krajMin);
  }
  sort(begin(casovi), end(casovi),
       [](const cas& a, const cas& b) {
```

```
        return krajCasa(a) < krajCasa(b);
      });
  int brojOdrzanihCasova = 1;
  int kraj = krajCasa(casovi[0]);
  for (int i = 1; i < n; i++)
    if (pocetakCasa(casovi[i]) >= kraj) {
      kraj = krajCasa(casovi[i]);
      brojOdrzanihCasova++;
    }
  cout << brojOdrzanihCasova << endl;

  return 0;
}
```

Задатак: Зли учитељ

Зли учитељ Нави одлучио је да се освети својим ученицима за ометање часа. Следећи домаћи задатак који им буде задао радиће се у паровима. Учитељ за сваког ученика зна колико сати му је потребно да уради свој део задатка, а време потребно пару ученика даураде задатак је збир времена та два ученика. Учитељ жели да одреди колико највише сати може да зада за израду домаћег задатка (зато што жели да прикрије своје зле намере и да делује фер) тако да бар један пар ученика не може да стигне да уради задатак, без обзира на то како се ученици поделе у парове.

Улаз: Са стандардног улаза се учитава паран број ученика n ($1 \leq n \leq 10^6$). Затим се учитава n природних бројева који представљају бројеве дана потребних за израду задатка сваког ученика.

Излаз: На стандардни излаз исписати један природан број који представља број дана који учитељ треба да постави као рок за израду задатка.

Пример 1

Улаз	*Излаз*
6 5 7 2 6 4 6	10

Пример 2

Улаз	*Излаз*
6 5 7 2 6 4 6	10

Решење

Потребно је одредити упаривање ученика такво да је највеће време израде домаћег задатка најмање могуће. Другим речима, у низу бројева, потребно је пронаћи оно упаривање које даје најмањи могући (у односу на сва упаривања) максимални збир пара елемената (у односу на све парове). Учитељ може ученицима да остави рок који је за један мањи од тог минималног максималног збира и биће сигуран да неће сви ученици моћи да на време заврше домаћи (и то ће бити најдужи такав рок).

Минимални максимални збир пара можемо одредити грамзивим алгоритмом. Сортирајмо времена потребна да се уради домаћи. Размотримо упаривање у коме је најспорији ученик a_{n-1} упарен са неким учеником a_i, али који није најбржи ($i > 0$). Тада је најбржи ученик a_0 упарен са неким учеником a_j, али који није најспорији ($j < n-1$). Ако сада направимо размену тако да упаримо ученике a_0 и a_{n-1} (најбржег и најспоријег) и ученике a_i и a_j, максимум се може само смањити. Остали парови остају непромењени, па је потребно посматрати само упаривања ова 4 ученика. Важи да је $\max(a_0 + a_j, a_i + a_{n-1}) = a_i + a_{n-1}$, јер је $a_0 \leq a_i$ и $a_j \leq a_{n-1}$. Пошто је $a_0 \leq a_i$, важи да је $a_0 + a_{n-1} \leq a_i + a_{n-1}$, а пошто је $a_j \leq a_{n-1}$, важи да је $a_i + a_j \leq a_i + a_{n-1}$. Дакле, оба броја $a_0 + a_{n-1}$ и $a_i + a_j$ су мања или једнака од вредности $a_i + a_{n-1} = \max(a_0 + a_j, a_i + a_{n-1})$, па важи $\max(a_0 + a_{n-1}, a_i + a_j) \leq \max(a_0 + a_j, a_i + a_{n-1})$. Према томе, постоји упаривање у ком се постиже минимални максимални збир, а у коме су најбржи и најспорији ученик упарени. Када се исти принцип примени на преостале ученике, закључујемо да је један начин да добијемо оптимално упаривање да се други најбржи упари са другим најспоријим, трећи најбржи са трећим најспоријим и тако даље. Овим смо у потпуности одредили упаривање које може дати минималну максималну разлику и њену вредност одређујемо испитивањем збирова времена свих парова ученика (након сортирања које се врши у времену $O(n \log n)$, то се може урадити у времену $O(n)$).

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```cpp
using namespace std;

int main() {
  // ucitavamo vremena potrebna ucenicima za izradu zadataka
  int n;
  cin >> n;
  vector<int> a(n);
  for(int i = 0; i < n; i++)
    cin >> a[i];

  // sortiramo vremena
  sort(begin(a), end(a));
  // minimalni maksimalni zbir se dobija u uparivanju u kojem je prvi
  // ucenik uparen sa poslednjim, drugi sa pretposlednjim itd.
  // odredjujemo maksimalni zbir nekog para u tom uparivanju
  int m = 0;
  for (int i = 0; i < n / 2; i++)
    m = max(m, a[i] + a[n - 1 - i]);

  // trazeno vreme je za jedan manje od tog minimuma
  cout << m - 1 << endl;

  return 0;
}
```

Задатак: Мали поштар

Јовица зарађује џепарац тако што доноси пакете својим комшијама. Поделу креће од своје куће и потребно је да пакете разнесе у друге куће распоређене дуж улице и да се врати назад у своју кућу. За сваку кућу познато је растојање од почетка улице. Најкраћи пут би прешао ако би пакете сложио тако да их редом дели комшијама дуж улице. Пошто Јовица жели да буде у доброј физичкој форми, он током поделе пакета трчи и жели да пакете уреди тако да пређе што већи пут. Напиши програм који одређује највећи пут који може да пређе.

Улаз: Са стандардног улаза се уноси број кућа у које треба донети пакете (међу њима се налази и Јовицина кућа), а затим и растојања тих кућа од почетка улице.

Излаз: На стандардни излаз исписати највеће растојање које Јовица може прећи током поделе пакета.

Пример 1

Улаз	*Излаз*
5 7 3 6 10 2	24

Објашњење

Постоји више начина да Јовица претрчи 24 дужне јединице. На пример, ако је његова кућа на позицији 3, он може да редом обилази куће 3, 7, 2, 10, 6, 3.

Пример 2

Улаз

```
7
3 5 11 4 2 17 9
```

Излаз

```
56
```

Решење

Решење грубом силом подразумева да се провере сви могући редоследи обиласка n кућа, тј. свих $n!$ пермутација датих бројева и да се утврди која од њих даје највећу могућу вредност пређеног пута. Пермутације се могу обићи коришћењем техника приказаних у задацима Следећа пермутација или Све пермутације. Алгоритам заснован на провери свих пермутација је изразито неефикасан и може се применити само на веома, веома мале улазе (практично, само за $n \leq 10$ програм може да реши задатак у датом временском ограничењу).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int zbirApsRazlika(const vector<int>& a) {
  int zbir = 0;
  for (size_t k = 1; k < a.size(); k++)
    zbir += abs(a[k] - a[k-1]);
  zbir += abs(a[a.size()-1] - a[0]);
  return zbir;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int maks = 0;
  do {
    maks = max(zbirApsRazlika(a), maks);
  } while(next_permutation(begin(a), end(a)));

  cout << maks << endl;
  return 0;
}
```

Интуитивно нам је јасно да ће се пуно претрчати ако се стално трчи са једног на други крај улице. Један грамзиви приступ је да се прво обиђе крајња лева кућа, па затим крајња десна, па друга слева, па друга здесна и тако "цик-цак". Докажимо да је ово грамзиво решење исправно.

За дати распоред $x_0, \ldots x_{n-1}$ одређујемо суму апсолутних вредности разлика елемената тј. покушавамо да максимизујемо суму

$$|x_0 - x_1| + |x_1 - x_2| + ... + |x_{n-2} - x_{n-1}| + |x_{n-1} - x_0|.$$

Сваки елемент се у суми јавља тачно два пута. У зависности од међусобног односа бројева неки елементи ће бити узети са знаком $+$, а неки са знаком $-$, и то тако да се тачно елемената узима са знаком $+$ и тачно елемената узима са знаком $-$. Циљ нам је да елементи који се узимају са знаком $+$ буду што већи, а да ови са знаком $-$ буду што мањи. Распоред који иде цик-цак постиже да се за знаком $+$ узме $\frac{n}{2}$ већих елемената низа, а са знаком $-$ узме $\frac{n}{2}$ мањих елемената низа (ако их је непаран број, тада се средњи узима једном са знаком $-$, а једном са знаком $+$). Заиста, ако имамо 6 елемената $a_0 \leq a_1 \leq a_2 \leq a_3 \leq a_4 \leq a_5$, цик-цак распоред даје вредност

$$|a_0 - a_5| + |a_5 - a_1| + |a_1 - a_4| + |a_4 - a_2| + |a_2 - a_3| + |a_3 - a_0|,$$

што је једнако

$$(a_5 - a_0) + (a_5 - a_1) + (a_4 - a_1) + (a_4 - a_2) + (a_3 - a_2) + (a_3 - a_0),$$

тј.

$$2 \cdot (a_5 + a_4 + a_3) - 2 \cdot (a_2 + a_1 + a_0)$$

Јасно је да се не може добити више од овога, а ово се, видели смо, увек може експлицитно достићи баш цик-цак распоредом.

У овом примеру смо коректност грамзиве стратегије доказали тако што смо израчунали теоријско ограничење функције која се оптимизује и затимо смо доказали да се грамзивом стратегијом достиже то теоријско ограничење.

Елементе низа можемо експлицитно сортирати, па затим распоредити елементе у нови низ по цик-цак редоследу и за тај нови низ израчунати збир апсолутних вредности разлика суседних елемената.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  sort(begin(a), end(a));
  vector<int> b(n);
  int i = 0, j = n-1;
  for (int k = 0; k < n; k++)
    if (k % 2 != 0)
      b[k] = a[i++];
    else
      b[k] = a[j--];

  int zbir = 0;
  for (int k = 1; k < n; k++)
    zbir += abs(b[k] - b[k-1]);
  zbir += abs(b[n-1] - b[0]);

  cout << zbir << endl;
  return 0;
}
```

Помоћни низ се може веома једноставно избећи у имплементацији.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  sort(begin(a), end(a));
  int i = 0, j = n-1;
```

```
  int zbir = 0;
  bool paran = true;
  while (i < j) {
    zbir += abs(a[j]-a[i]);
    if (paran)
      i++;
    else
      j--;
    paran = !paran;
  }
  zbir += abs(a[0] - a[i]);
  cout << zbir << endl;
  return 0;
}
```

Ако пажљиво размотримо доказ коректности, примећујемо да распоред у коме идемо од прве до последње, па до друге, затим претпоследње куће итд., није једини који даје максимални пређени пут. Довољно је само да наизменично узимамо елементе из прве и друге половине низа (у било ком редоследу). Ово инспирише још једноставнији алгоритам за израчунавање траженог максимума (саберемо $\frac{n}{2}$ бројева са почетка, одузмемо $\frac{n}{2}$ бројева са краја низа и помножимо са 2).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  sort(begin(a), end(a));

  int zbir = 0;
  for (int i = 0; i < n/2; i++) {
    zbir += a[n-1-i];
    zbir -= a[i];
  }

  cout << zbir * 2 << endl;
}
```

Задатак: Минимална сума два броја формирана од датих цифара

Написати програм који одређује минималну могућу суму два броја формирана од цифара датог низа (од 0 до 9). Сваки елемент низа мора бити употребљен било у једном, било у другом броју, а бројеви могу да почну и нулом. На пример, за цифре 5, 3, 0, 7, 5 добија се најмања сума 92 (јер је $35 + 057 = 92$).

Улаз: Прва линија стандардног улаза садржи број цифара n ($2 \leq n \leq 30$), а следећих n линија по једну цифру.

Излаз: Исписати у једној линији стандардног излаза тражену минималну суму.

Пример 1

Улаз	*Излаз*
5	92
5	
3	
0	
7	
5	

Пример 2

Улаз	*Излаз*
6	381
1	
2	
3	
4	
5	
6	

Пример 3

Улаз	*Излаз*
5	3
0	
1	
0	
2	
0	

Задатак: Најмањи број надовезивањем више бројева

Дат је низ бројева $a_i, i = 0, 1, \cdots, n-1$. Под надовезивањем (конкатенацијом) два броја x и y подразумевамо број xy, који се добија надовезивањем цифара броја y после броја x (нпр. надовезивањем бројева 123 и 45 добија се број 12345). Одредити најмањи број који се може добити надовезивањем свих бројева низа a_i, $i = 0, 1, \cdots, n-1$.

Улаз: У првој линији стандардног улаза наведен је број елемената низа n ($2 \leq n \leq 1000$), а у следећих n линија по један природан број између 1 и 10^6.

Излаз: Исписати у једној линији стандардног излаза тражени најмањи број.

Пример 1

Улаз	*Излаз*
5	1112323987
32	
11	
987	
12	
3	

Пример 2

Улаз	*Излаз*
2	91919191919
91919	
919191	

Решење

С обзиром на величину резултујућег броја и природу проблема, јасно је да бројеве не можемо представити уобичајеним (ограниченим) типовима за представљање целих бројева. Показаће се да се бројеви могу представити нискама карактера. Међу тим нискама можемо успоставити одређени поредак, такав да се најмањи резултујући број добија када се низ ниски сортира у неопадајућем поретку.

Надовезивањем свих ниски у би ком редоследу добија се резултујућа ниска са истим бројем цифара. Поређење два броја са истим бројем цифара се подудара са њиховим лексикографским поређењем. Стога је довољно надовезане ниске поредити лексикографски.

Централно питање је како одредити поредак две ниске (сортирање више ниски се онда врши у односу на тај поредак). Јасно је да приликом надовезивања прво треба да ставимо ону ниску која има мању прву цифру, да ако су прве цифре једнаке треба да гледамо другу цифру и да прво надовезујемо број са мањом другом цифром итд. Нпр. ако се надовезују бројеви 235 и 2462, прво треба ставити први број. Делује, дакле, да је ниске довољно поредити лексикографски, међутим, то није увек случај. На пример, ниска 35 је лексикографски мања од 352, али је боље надовезати 35235 од 35352. Када је једна ниска префикс друге, тада се поредак одређује поређењем прве (заједничке) цифре обе ниске и прве цифре дуже ниске иза заједничког префикса (у овом примеру поредили смо 3 и 2). Ако су оне једнаке пореде се наредне цифре и тако даље (дакле, суштински се опет врши лексикографско поређење). Интересантан је случај односа ниски 121 и 12, где је ниска 12112 мања од 12121 и односа веома сличних ниски 212 и 21, где је, обратно него у претходном случају ниска 21212 мања од ниске 21221. Поређење, дакле, није довољно завршити када се дође до краја обе ниске, већ је заправо најбоље ово поређење тумачити тако што се лексикографски пореде обе ниске настале надовезивањем полазних у различитим редоследима (за полазне ниске a и b, лексикографски се пореде ниске ab и ba).

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
```

```
// poredjenje niski - rezultat poretka je odredjena vrednoscu
// konkatenacije
bool poredi(const string& a, const string& b) {
  return a + b < b + a;
}

int main() {
  // broj clanova niza
  int n;
  cin >> n;
  // ucitavamo elemente niza a
  vector<string> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  // sortiramo niz
  sort(begin(a), end(a), poredi);
  // ispisujemo rezultat
  for (int i = 0; i < n; i++)
    cout << a[i];
  cout << endl;
}
```

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

// poredjenje niski - rezultat poretka je odredjena vrednoscu
// konkatenacije
bool poredi(const string& a, const string& b) {
  return a + b < b + a;
}

int main() {
  // broj clanova niza
  int n;
  cin >> n;
  // ucitavamo elemente niza a
  vector<string> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  // sortiramo niz
  sort(begin(a), end(a), poredi);
  // ispisujemo rezultat
  for (int i = 0; i < n; i++)
    cout << a[i];
  cout << endl;
}
```

Мало неефикаснији, али за реализацију доста једноставнији начин да се ово поређење реализује је да се експлицитно изврше оба могућа надовезивања и да се види које од њих даје лексикографски мању ниску коришћењем библиотечког лексикографског поређења ниски.

Задатак: Излог са два реда књига

Продавачица жели да уреди излог своје књижаре. Књиге које има ће су поређане у два реда од по n књига и тако треба и да остану (није могуће премештати књиге из једног у други ред). За сваку књигу је позната боја

и висина. И у једном и у другом реду књиге морају бити сложене по унаред задатом редоследу боја (тако да кодови боја буду поређани неопадајуће). Књиге исте боје се могу ређати у произвољном редоследу. Такође, свака књига у предњем реду мора бити нижа него књига иза ње (да би се обе књиге виделе). Напиши програм који помаже да се одреди редослед књига или констатује да редослед који задовољава тражене услове није могуће направити.

Улаз: Прва линија стандардног улаза садржи природан број n ($1 \leq n \leq 5 \cdot 10^5$). У наредне четири линије налази се по n природних бројева (сваки између 1 и 10^9, раздвојених размацима). У првом реду се налазе ознаке боја књига у задњем реду, у наредном висине књига у задњем реду (у односу на тренутни распоред књига у задњем реду). У наредном реду се налазе ознаке боја књига у предњем реду и у наредном висине књига у предњем реду (у односу на тренутни распоред књига у предњем реду).

Излаз: На стандардни излаз исписати две линије са бројевима од 1 до n раздвојених размацима који представљају пермутацију књига у задњем реду и пермутацију књига у предњем реду која одређује распоред који задовољава дате услове (на месту i налази се број књиге у полазном распореду која треба да дође на то место). Ако књиге није могуће разместити у траженом распореду, исписати -.

Пример 1

Улаз	*Излаз*
4	3 2 4 1
3 2 1 2	4 2 1 3
2 3 4 3	
2 1 2 1	
2 2 1 3	

Објашњење

Распоред након рамештања је:

```
boje gore:   1 2 2 3
visine gore: 4 3 3 2
boje dole:   1 1 2 2
visine dole: 3 2 2 1
```

Пример 2

Улаз

```
2
1 2
2 3
2 8
2 1
```

Излаз

```
-
```

Задатак: Неопадајућа растојања суседних елемената

Напиши програм који елементе датог низа распоређује тако да се након израчунавања низа апсолутних разлика суседних елемената добијеног низа добија неопадајући низ.

Улаз: Са стандардног улаза се учитава дужина низа n ($1 \leq n \leq 50000$), а затим и елементи низа (цели бројеви између -10^9 и 10^9, сваки у посебном реду).

Излаз: На стандардни излаз исписати елементе низа у редоследу добијеном након траженог распоређивања, раздвојене са по једним размаком. Ако постоји више таквих распореда, исписати било који.

Пример

Улаз	*Излаз*
6 3 -2 4 5 10 -8	4 3 -2 5 -8 10

Задатак: Гласници

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Грамзиво решење

Задатак има веома елегантно грамзиво решење, које је директно, тј. не користи претрагу. Једна оптимална стратегија је да сви гласници који знају поруку трче на десно према онима који не знају поруку, а да они који не знају поруку трче на лево, према онима који знају поруку ако већ нису у њиховом домету, а на десно ако јесу у њиховом домету. При том, гласници који трче на десно све време вичу, тако да се порука сазна чим неки гласник уђе у њихов домет (што је домет најдешњег од њих). У таквој стратегији сваки гласник сазнаје поруку од свог непосредног претходника.

Претпоставимо да је t_i најмање могуће време да гласник i сазна поруку и да је најдаља могућа позиција у којој се он тада налази једнака G_i. Одредимо ове вредности индуктивном конструкцијом.

- За првог гласника важи да је $t_0 = 0$ и да је $G_0 = g_0$.
- Одређујемо ове податке за наредног гласника t_{i+1} и G_{i+1}.

  Ако се наредни гласник у времену t_i може наћи у домету гласника i, тада је $t_{i+1} = t_i$. Једини начин да се то не догоди је да је $G_i + d < g_{i+1} - t_i$. У том случају је потребно време продужити тако да се гласник i још мало креће на десно, а гласник $i + 1$ још мало креће на лево. С обзиром да је брзина јединична, време након ког порука може бити пренета је $t_{i+1} = t_i + \frac{(g_{i+1} - t_i) - (G_i + d)}{2}$.

  Вредност G_{i+1} тј. најдаља тачка на којој гласник $i + 1$ у тренутку t_{i+1} може да прими поруку ће бити десна граница домета $G_i + d$ у свим случајевима, осим ако је гласник $i + 1$ превише лево и не може за време t_{i+1} стићи до ње. Зато важи даје $G_{i+1} = \min(G_i + d, g_{i+1} + t_{i+1})$.

Пошто се оптимално решење конструише једним проласком кроз низ гласника, сложеност овог решења је $O(n)$.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  double domet;
  cin >> domet;
  int n;
  cin >> n;
  vector<double> pozicije(n);
  for (int i = 0; i < n; i++)
    cin >> pozicije[i];

  double t = 0.0;
```

```
  double p = pozicije[0];
  for (int i = 1; i < n; i++) {
    if (pozicije[i] - t - p > domet) {
      t += (pozicije[i] - t - p - domet) / 2.0;
      p = pozicije[i] - t;
    } else {
      p = min(pozicije[i] + t, p + domet);
    }
  }

  cout << fixed << showpoint << setprecision(3)
       << t << endl;

  return 0;
}
```

Задатак: Мост

Група људи током ноћи стоји на једној обали реке. Имају само једну светиљку. Преко реке је разапет мост преко ког истовремено могу да пређу највише две особе и то само ако носе светиљку. За сваку особу је познато време потребно за прелазак моста. Ако две особе прелазе мост заједно, крећу се брзином спорије особе. Напиши програм који одређује најмање време потребно да сви људи из групе пређу на другу обалу реке.

Улаз: Са стандардног улаза се учитава број особа n ($1 \leq n \leq 100$), а затим, у n наредних редова времена потребна да тих n особа пређе мост.

Излаз: На стандардни излаз исписати најмање време потребно да сви пређу на другу обалу.

Пример

Улаз	*Излаз*
4 1 2 5 10	17

Објашњење

Прво прелазе особа 1 и 2 и враћа се особа 1 (за то им је потребно $\max(1, 2) + 1 = 3$ минуте). Након тога заједно прелазе особа 3 и 4 и враћа се особа 2 (за то им је потребно $\max(5, 10) + 2 = 12$ минута). Након тога заједно прелазе особа 1 и 2 (за то им је потребно $\max(1, 2) = 2$ минуте). Укупно је, дакле, потребно $3 + 12 + 2 = 17$ минута.

Решење

Груба сила - динамичко програмирање

Решење грубом силом подразумева да се у сваком кораку размотре све могућности за прелаз на другу обалу. Притом са почетне обале увек пребацујемо две особе, а на почетну враћамо једну особу (тако се након сваког преласка и враћања број људи који су пребачени на другу обалу бива увећан). Прелазак само једне особе нема смисла, јер након враћања светиљке долазимо у исто стање као пре њеног преласка или евентуално у стање где је једна особа замењена другом. Ако смо бржу особу на полазној обали заменили споријом, само смо умањили шансе да обавимо брз прелазак (јер бржа особа може да ради све што може и спорија и то брже), а ако смо спорију особу заменили бржом, знамо да смо у претходним прелазима могли бити ефикаснији да је у самом почетку уместо те спорије особе на другу обалу прешла бржа. Враћање две особе такође нема смисла, јер након тога долазимо у конфигурацију у којој смо могли бити да се једна од те две особе није шетала са једне обале на другу и назад (што је само могло да увећа укупно време).

После сваког преласка и враћања рекурзивно решавамо проблем истог облика, али мање димензије. Рекурзивна функција решава проблем у ком се светиљка налази на полазној обали и анализира све могућности за

прелазак светиљке на другу обалу и враћање светиљке назад (након чега се рекурзивно решава проблем из стања у које се на тај начин дошло). Параметар који се мења током рекурзије је скуп особа које се налазе са једне и скуп особа које се налазе са друге стране реке. Излаз из рекурзије је случај када су на почетној обали остале само једна или две особе и они тада заједно прелазе реку завршавајући прелаз.

Пошто се током ове исцрпне претраге много пута долази у исто стање и рекурзивни позиви се понављају. Имплементација се, дакле, може убрзати тако што се примене технике динамичког програмирања (најједноставније је имплементирати мемоизацију). За сваки подскуп скупа особа на полазној обали, у низу памтимо резултат рекурзивног позива тј. најмање време потребно да све особе из тог подскупа пређу на другу обалу (када је светиљка код њих). Сваки такав подскуп можемо кодирати битовима неозначеног целог броја.

Сложеност овог поступка (и уз мемоизацију) је експоненцијална јер postoji 2^n подскупова скупа од n особа (већ за скупове са око 15 особа, функција постаје прилично неефикасна).

Иако постоје начини да се ово решење убрза (на пример, може се приметити да увек треба вратити најбржу пребачену особу), нећемо настављати развој програма у овом смеру, зато што се другачијим приступом могу добити много ефикаснија и једноставнија решења.

Оптимална стратегија - грамзиви приступ

Ефикасно решење можемо постићи изналажењем стратегије преласка за који се унапред може доказати да даје оптимално време преласка. Стратегија ће бити заснована на томе да се (грамзиво) најспорија особа пребацује на другу обалу, одакле се даље не помера. Циљ нам је, дакле, да мало по мало повећавамо скуп особа које су трајно премештене. Прилично је јасно да споре особе треба да буду те чије се шетање са једне на другу обалу мора избећи. Најспорија особа ће кад тад морати бити премештена на долазну обалу, када се једном премести нема разлога да се враћа назад (уместо ње, увек је боље да се враћају брже особе). Пошто се најспорија особа мора кад тад пребацити, нема разлога да се то не уради одмах и да се та особа у потпуности избаци из даљег разматрања. Када се након тога лампа врати назад, редукује се димензија проблема и алгоритам се даље наставља по истом принципу.

Претпоставимо да су времена потребна за прелазак сортирана и дата кроз низ $v_0 \leq v_1 \leq v_2 \ldots \leq v_{k-1}$.

У случају када су на полазној обали највише две особе се решава директно, тако што те особе прелазе на другу обалу (ако једина преостала особа i прелази сама, потребно је време v_i, а ако су преостале особе $i < j$, потребно је време v_j). У супротном покушавамо да пребацимо две најспорије особе и да вратимо светљку на полазну обалу.

Посебно ћемо размотрити и случај три особе, јер се показује да је и он другачији од општег случаја у коме има бар 4 особе. У том случају лако се види да је најбоље решење да најбржа особа преводи једну по једну од две спорије особе (чиме је време преласка једнако збиру времена те три особе $v_0 + v_1 + v_2$). Мост се не може прећи за мање од овог времена, јер ако би две спорије особе ишле заједно, тада би једна од њих морала да се врати по најбржу па би време било или $2v_1 + v_2$ или $2v_2 + v_1$, што је спорије од $v_0 + v_1 + v_2$.

Претпоставимо сада да на полазној обали постоје бар 4 особе и размотримо ефикасне начине да се најспорија особа преведе на другу обалу.

- Једна могућност је да најспорија особа истовремено пређе реку са другом најспоријом особом и да се ниједна од њих више не враћа назад. Наиме, за прелазак најспорије особе већ је потребно време v_{k-1} и за то време се може истовремено превести и особа v_{k-2}, чиме се постиже то да се време v_{k-2} неће посебно трошити. Ако две најспорије особе пређу заједно, проблем је то што неко од њих мора да врати назад светиљку, а то није ефикасно. Зато је пожељно обезбедити да на другој обали буде неко ко ће вратити светиљку када најспорије две особе заједно пређу мост. То не сме бити ниједна од раније преведених особа (јер смо за њих, као веома споре особе, констатовали да нема сврхе да се враћају назад те да се оне неће померати и да их можемо у потпуности изоставити из разматрања) и морамо ангажовати неког од преосталих особа да то уради. Наравно, те особе треба да буду што брже. Морамо прво пребацити две најбрже особе, затим вратити једну од њих, заједно са светиљком, затим пребацити две најспорије заједно и затим употребити ону другу најбржу особу која и даље чека на супротној обали да врати светиљку. Време потребно да се ово оствари је $v_{k-1} + v_0 + 2v_1$.
- Друга могућност да је да неко преведе најспорију особу и да затим врати светиљку. Да бисмо могли да упоредимо однос ове и претходне могућности, размотримо потезе који би нас довели у исту ситуацију као тамо, тј. размотримо потезе који би довели до тога да су две најспорије особе преведене и да је светиљка враћена. Ако не иду две најспорије особе заједно, тада је најбоље да, слично као у случају три

особе, неко преведе једну од две најспорије особе, да затим врати светиљку, затим да неко преведе другу од те две најспорије особе и да затим поново врати светиљку. У оба преласка требало би да учествује што бржа особа (да би што пре вратила светиљку), па је најбоље да оба пута светиљку враћа најбржа особа. Дакле, ако је v_0 време потребно да најбржа особа пређе мост, а ако су v_{k-2} и v_{k-1} времена потребна да две најспорије особе пређу мост, укупно време потребно да се оне пребаце и светиљка врати биће $v_{k-2} + v_{k-1} + 2v_0$.

Након оба приказана сценарија две најспорије особе су пребачене, ниједна од њих се више неће враћати, док је светиљка враћена назад и проблем је сведен на исти облик и две особе мање. Који од два могућа сценарија се користи зависи од тога који захтева мање времена (питање је односа величина $v_0 + v_{k-2}$ и $2v_1$).

Пример у коме је прва стратегија оптимална може бити онај у коме је особама потребно 1, 2, 8 и 10 минута, јер удруживањем особа 3 и 4 избегавамо посебно шетање у трајању од 8 минута (што плаћамо додатним преласцима прве две особе, но то се исплати, јер је њима за прелазак потребно јако мало времена). Заиста, важи да је $1 + 8 > 2 \cdot 2$.

Пример у коме је друга стратегија оптимална може бити онај у коме је особама потребно 1, 4, 5 и 6 минута. Уштеда 5 минута код треће особе није вредна додатних прелазака друге особе којој је потребно 4 минута. Заиста, важи да је $1 + 5 < 2 \cdot 4$.

Може још да се постави питање зашто није узет у обзир сценарио у коме најбржа особа преводи најспорију, враћа лампу, а затим не преводи другу по реду најспорију особу, већ та особа прелази заједно са трећом по реду најспоријом (јер се тако добија боље време). Да би се то догодило мора да важи да је $v_0 + v_{k-2} \leq 2v_1$, а да је $v_0 + v_{k-3} > 2v_1$, што није могуће, јер је $v_{k-3} \leq v_{k-2}$. Стога је оптимална стратегија таква да се групишу две по две најспорије особе које заједно прелазе мост, све до једног тренутка у ком то престане да буде исплативо и тада најбржа особа креће да преводи једну по једну особу све док сви не пређу.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  // broj osoba na levoj strani
  int n;
  cin >> n;
  vector<int> osobe(n);
  for (int i = 0; i < n; i++)
    cin >> osobe[i];
  // sortiramo osobe rastuce po potrebnom vremenu prelaska
  sort(begin(osobe), end(osobe));

  // ukupno potrebno vreme
  int vreme = 0;
  // dok još ima osoba na levoj strani
  while (n > 0) {
    if (n == 1) {
      // ostala je jedna osoba i ona prelazi sama
      vreme += osobe[0];
      n -= 1;
    } else if (n == 2) {
      // ostale su dve osobe i one prelaze zajedno
      vreme += osobe[1];
      n -= 2;
    } else if (n == 3) {
      // ostale su tri osobe o0 <= o1 <= o2
      // odlaze o0 i o1 za vreme o1,
      // vraca se o0 za vreme o0,
      // odlaze o0 i o2 za vreme o2 - ukupno vreme o0 + o1 + o2
      vreme += osobe[0] + osobe[1] + osobe[2];
```

```
      n -= 3;
    } else {
      // premestamo dve najsporije osobe na drugu stranu
      // neka je o0 <= o1 <= ... <= m2 <= m1
      // imamo dve mogucnosti i uzimamo bolju od njih:

      //   odlaze o0 i o1 za vreme o1      o0 o1      m1 m2
      //   vraca se o0 za vreme o0         o1         o0 m1 m2
      //   odlaze m2 i m1 za vreme m1      o1 m1 m2   o0
      //   vraca se o1 za vreme o1         m1 m2      o0 o1
      //   ukupno vreme: o0 + 2*o1 + m1

      //   odlaze o0 i m2 za vreme m2      o0 m2      o1 m1
      //   vraca se o0 za vreme o0         m2         o0 01 m1
      //   odlaze o0 i m1 za vreme m1      o0 m1 m2   o1
      //   vraca se o0 za vreme o0         m1 m2      o0 01
      //   ukupno vreme 2*o0 + m1 + m2

      int vreme1 = osobe[0] + 2*osobe[1] + osobe[n-1];
      int vreme2 = 2*osobe[0] + osobe[n-2] + osobe[n-1];
      vreme += min(vreme1, vreme2);

      // uklanjamo dve najsporije osobe iz reda osoba sa leve strane
      n -= 2;
    }
  }

  // ispisujemo ukupno vreme
  cout << vreme << endl;
  return 0;
}
```

Глава 9

Структуре података

9.1 Стек

Стек представља колекцију података у коју се подаци додају по LIFO (енгл. last-in-first out) принципу - елемент се може додати и скинути само на врха стека.

У језику C++ стек се реализује класом `stack<T>` где `T` представља тип елемената на стеку. Подржане су следеће методе:

- `push` - поставља дати елемент на врх стека
- `pop` - скида елемент са врха стека
- `top` - очитава елемент на врху стека (под претпоставком да стек није празан)
- `empty` - проверава да ли је стек празан
- `size` - враћа број елемената на стеку

Стек у језику C++ је заправо само адаптер око неке колекције података (подразумеано вектора) који корисника тера да поштује правила приступа стеку и спречава да направи операцију која над стеком није допуштена (попут приступа неком елементу испод врха).

Задатак: Линије у обратном редоследу

Напиши програм који исписује све линије које се учитају са стандардног улаза у обратном редоследу од редоследа учитавања.

Улаз: Са стандардног улаза се учитавају линије текста, све до краја улаза.

Излаз: На стандардни излаз исписати учитане линије у обратном редоследу.

Пример

Улаз	*Излаз*
`zdravo`	`dan`
`svete`	`dobar`
`dobar`	`svete`
`dan`	`zdravo`

Решење

Једно од могућих решења је да се све учитане линије сместе на стек, а да се затим испишу узимајући једну по једну са стека. Пошто стек функционише по принципу LIFO (last in first out, тј. онај који последњи уђе, први излази), редослед ће бити обрнут (најкасније додата линија биће прва скинута и исписана, док ће прва постављена линија бити скинута и исписана последња).

```
#include <iostream>
#include <string>
#include <stack>

using namespace std;
```

```
int main() {
  stack<string> s;
  string linija;
  while (cin >> linija)
    s.push(linija);
  while (!s.empty()) {
    cout << s.top() << endl;
    s.pop();
  }
  return 0;
}
```

Задатак: Прегледање веба

Прегледач веба памти историју посећених сајтова и корисник има могућност да се враћа унатраг на сајтове које је раније посетио. Написати програм који симулира историју прегледача тако што се учитавају адресе посећених сајтова (свака у посебном реду), а када се учита ред у коме пише `back` прегледач се враћа на последњу посећену страницу. Ако се наредбом `back` вратимо на почетну страницу, исписати `-`. Ако се на почетној страници изда наредба `back`, остаје се на почетној страници. Програм треба да испише све сајтове које је корисник посетио.

Улаз: Са стандардног улаза се учитавају веб-адресе, свака у посебној линији.

Излаз: На стандардни излаз исписати редом сајтове који се посећују.

Пример

Улаз	*Излаз*
`http://www.google.com`	`http://www.google.com`
`http://www.rts.rs`	`http://www.rts.rs`
`back`	`http://www.google.com`
`http://www.petlja.org`	`http://www.petlja.org`
`http://www.matf.bg.ac.rs`	`http://www.matf.bg.ac.rs`
`back`	`http://www.petlja.org`
`back`	`http://www.google.com`
`back`	`-`
`back`	`-`

Задатак: Сажимање путања

Путања у систему Linux се састоји од назива директоријума раздвојених карактером `/`, која може да се заврши именом директоријума или датотеке. Претпоставићемо да се Имена директоријума и датотека састоје од малих слова енглеске абецеде, цифара, подвлака и карактера `.`. Специјални директоријуми у путањи су `.` који означава текући директоријум и `..` који означава родитељски директоријум (родитељским директоријумом кореног директоријума сматра се он сам). Сажимање путање подразумева да се из њеног назива уклоне ти специјални директоријуми.

Улаз: Са стандардног улаза се учитава исправна путања која почиње карактером `/`.

Излаз: На стандардни излаз се исписује сажета путања.

Пример 1

Улаз	*Излаз*
`/abc/./def/../jkl/../mnp/doc.txt`	`/abc/mnp/doc.txt`

Пример 2

Улаз	*Излаз*
`/abc/../../xyz.jpeg`	`/xyz.jpeg`

Задатак: Упареност заграда

У изразу учествују следеће врсте заграда `(`, `)`, `{`, `}`, `[` и `]`. Напиши програм који проверава да ли су у унетом изразу заграде исправно упарене.

Улаз: Са стандардног улаза се уноси израз (цео у једном реду, без размака).

Излаз: На стандардни излаз исписати `da` ако су заграде исправно упарене тј. `ne` ако нису.

Пример 1

Улаз	*Излаз*
`[3*(2+4)]*5`	`da`

Пример 2

Улаз	*Излаз*
`{3*(2+4)}*[5+7)+(4*5)`	`ne`

Задатак: Push-pop реконструкција

Током рада са стеком укупно n пута је извршена операција `push` којом се нека вредност поставља на врх стека и укупно n пута је извршена операција `pop` којом је елемент скинут са врха стека. Ако је познат низ бројева који су редом били аргументи операције `push` и низ бројева који су редом добијани као резултат операције `pop`, напиши програм који одређује редослед операција `push` и `pop`.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим два низа од по n бројева раздвојених размацима. Претпоставити да су и у једном и у другом низу сви елементи различити.

Излаз: На стандардни излаз исписати редослед операција `push` и `pop` или `-`, ако такав редослед операција није могуће пронаћи за задате низове.

Пример 1

Улаз	*Излаз*
`5`	`push`
`1 2 3 4 5`	`push`
`5 4 3 2 1`	`push`
	`push`
	`push`
	`pop`
	`pop`
	`pop`
	`pop`
	`pop`

Пример 2

Улаз	*Излаз*
`5`	`push`
`1 2 3 4 5`	`push`
`3 2 5 4 1`	`push`
	`pop`
	`pop`
	`push`
	`push`
	`pop`
	`pop`
	`pop`

Пример 3

Улаз	*Излаз*
`5`	`-`
`1 2 3 4 5`	
`5 4 3 1 2`	

Задатак: Сажимање серија узастопних једнаких елемената

Низ се трансформише тако што се сваких k или више узастопних појављивања неког елемента бришу. Напиши програм који одређује садржај низа након исцрпне примене ове трансформације.

Улаз: Са стандардног улаза се учитава број k ($1 \leq k \leq 10^4$), затим број n ($1 \leq n \leq 10^6$) и на крају елементи низа одвојени размаком.

Излаз: На стандардни излаз исписати елементе резултујућег низа, раздвојене размаком.

Пример

Улаз	*Излаз*
`3`	`3 3 2`
`20`	
`1 1 2 2 2 2 1 3 4 4 5 5 5 4 4 3 2 1 1 1`	

Објашњење

Након уклањања четири двојке, три јединице постају узастопне и оне се уклањају. Након укањања три петице, четири четворке постају узастопне и оне се уклањају. На крају се уклањају три узастопне јединице.

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Нерекурзивно брзо сортирање

Једна веома важна употреба стека је за реализацију рекурзије. Током извршавања рекурзивних функција, на стек се смештају вредности локалних променљивих и аргумената сваког активног позива функције. Рекурзију увек можемо уклонити и уместо системског стека можемо ручно одржавати стек са тим подацима.

Прикажимо ову технику уклањања рекурзије на примеру нерекурзивне имплементације алгоритма брзог сортирања. Нагласимо да је код алгоритма QuickSort дубина рекурзије мала (она логаритамски зависи од броја елемената низа), па се њеном елиминацијом не добија ништа значајно.

На стеку ћемо чувати аргументе рекурзивних позива функције сортирања. На почетку је то пар индекса $(0, n1)$. Главна петља се извршава све док се стек не испразни и у њој се обрађује пар индекса који се скида са врха стека. Уместо рекукрзивних позива њихове ћемо аргументе постављати на врх стека и чекати да они буду обрађени у некој од наредних итерација петље. Приметимо да се аргументи другог рекурзивног позива обрађују тек када се у потпуности реши потпроблем који одговара првом рекурзивном позиву, што одговара понашању функције када је заиста имплементирана рекурзивно.

Рецимо и да је ова техника општа и да се рекурзија увек може елиминисати на овај начин. За разлику од тога, елиминисање специфичних облика рекурзије (попут, на пример, репне) није увек применљиво, али када јесте, доводи до боље меморијске (па и временске) ефикасности јер се не користи стек.

```
void quick_sort(ctor<int>& a) {
  // stek na kome čuvamo argumente rekurzivnih poziva
  stack<pair<int, int>> sortirati;
  // sortiranje kreće od obrade celog niza tj. pozicija (0, n-1)
  sortirati.emplace(0, n - 1);
  while (!sortirati.empty()) {
    // skidamo par (l, d) sa vrha steka
    auto p = sortirati.top();
    int l = p.first, d = p.second;
    sortirati.pop();
    // obrađujemo par (l, d) na isti način kao u rekurzivnoj implementaciji
    if (d - l < 1)
      continue;
    int k = l;
    for (int i = l+1; i <= d; i++)
      if (a[i] < a[l])
        swap(a[++k], a[i]);
    swap(a[k], a[l]);
    // umesto rekurzivnih poziva njihove argumente
    // postavljamo na stek
    sortirati.emplace(l, k-1);
    sortirati.emplace(k+1, d);
  }
}
```

Види другачија решења овог задатка.

Задатак: Број белих области

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Претрагу у дубину је могуће имплементирати и уз помоћ стека.

```
#include <iostream>
#include <stack>
#include <utility>
using namespace std;

// maksimalne dimenzije matrice
const int N = 20, M = 20;

// oznake polja u matrici
const int BELA = 0, CRNA = 1, OBELEZENO = -1;
```

```
// provera da li se polje (x, y) nalazi u matrici dimenzije mxn
bool UMatrici(int x, int y, int m, int n) {
  return x >= 0 && x < m && y >= 0 && y < n;
}

// obilazi se belo, neobelezeno polje sa koordinatama (x, y)
void Obelezi(int x0, int y0, int a[N][M], int m, int n) {
  stack<pair<int, int>> obeleziti;
  obeleziti.emplace(x0, y0);
  while (!obeleziti.empty()) {
    auto p = obeleziti.top(); obeleziti.pop();
    int x = p.first, y = p.second;
    // obelezavamo polje
    a[x][y] = OBELEZENO;
    // obilazimo sve susede
    int dx[] = { -1, 0, 1, 0 };
    int dy[] = { 0, 1, 0, -1 };
    for (int i = 0; i < 4; i++) {
      int xx = x + dx[i], yy = y + dy[i];
      // ako je susedno polje belo (i neobelezeno), obelezavamo ga
      if (UMatrici(xx, yy, m, n) && a[xx][yy] == BELA)
        obeleziti.emplace(xx, yy);
    }
  }
}

int main() {
  // ucitavamo matricu
  int m, n;
  cin >> m >> n;
  int a[M][N];
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      cin >> a[i][j];

  int oblast = 0;   // broj obelezenih oblasti
  // obilazimo sva polja u matrici
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      if (a[i][j] == BELA) {
        Obelezi(i, j, a, m, n);
        oblast++;
      }

  cout << oblast << endl;
}
```

Задатак: Minesweeper отварање

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Највећа заражена област

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: P-Q коњи

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Вредност постфиксног израза

Постфиксна нотација се понекад назива и пољска нотација, а постфиксна нотација се понекад назива и обратна пољска нотација (енгл. reverse polish notation, RPN) у част логичара Јана Лукашијевича који ју је изумео. Она подразумева да се бинарни оператори уместо између операнада записују након њих. На пример, уместо `3 + 5`, писаћемо `3 5 +`. Напиши програм који одређује вредност постфиксно записаног израза.

Улаз: Са стандардног улаза се учитава постфиксно записан израз који садржи једноцифрене бројеве и операторе `+` и `*` (без размака).

Излаз: На стандардни излаз исписати вредност учитаног израза.

Пример 1

Улаз	*Излаз*
`12+3*`	`9`

Објашњење

Постфиксно је записан израз `(1+2)*3`.

Пример 2

Улаз

```
11+2*345+*+
```

Излаз

```
31
```

Објашњење

Постфиксно је записан израз `(1+1)*2+3*(4+5)`.

Задатак: Превођење потпуно заграђеног израза у постфиксни облик

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Решење техником рекурзивног спуста се може прерадити тако да се добије нерекурзивна имплементација. Да бисмо се ослободили рекурзије, потребно је да употребимо стек. Кључна опаска је да се у стек оквиру функције, пре рекурзивног позива за превођење другог операнда памти оператор. Ово нам сугерише да нам је за нерекурзивну имплементацију неопходно да одржавамо стек на који ћемо смештати операторе. Када наиђемо на број преписујемо га на излаз, када наиђемо на оператор стављамо га на стек, а када наиђемо на затворену заграду скидамо и исписујемо оператор са врха стека.

Размотримо, на пример, израз `((3+4)*(5+2))`

- Први карактер је отворена заграда коју прескачемо.
- Наредни карактер је отворена заграда коју прескачемо.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `3`).
- Наредни карактер је оператор `+`, који постављамо на стек. Стек је сада `+`.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `34`).
- Наредни карактер је затворена заграда, па исписујемо оператор са врха стека (до сада је исписано `34+`). Стек је сада празан.
- Наредни карактер је оператор `*` који постављамо на стек. Стек је сада `*`.

- Наредни карактер је отворена заграда коју прескачемо.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `34+5`)
- Наредни карактер је оператор `+` који постављамо на стек. Стек је сада `*+`.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `34+52`)
- Наредни карактер је затворена заграда, па исписујемо оператор са врха стека (до сада је исписано `34+52+`). Стек је сада `*`.
- Наредни карактер је затворена заграда, па исписујемо оператор са врха стека (до сада је исписано `34+52+*`). Стек је сада празан.

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

bool jeOperator(char c) {
  return c == '+' || c == '*';
}

string prevedi(const string& izraz) {
  string postfiks;
  stack<char> operatori;
  for (char c : izraz) {
    if (isdigit(c))
      postfiks += c;
    else if (c == ')') {
      postfiks += operatori.top();
      operatori.pop();
    } else if (jeOperator(c))
      operatori.push(c);
  }
  return postfiks;
}

int main() {
  string izraz;
  cin >> izraz;
  string postfiks = prevedi(izraz);
  cout << postfiks << endl;
  return 0;
}
```

Задатак: Вредност израза

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

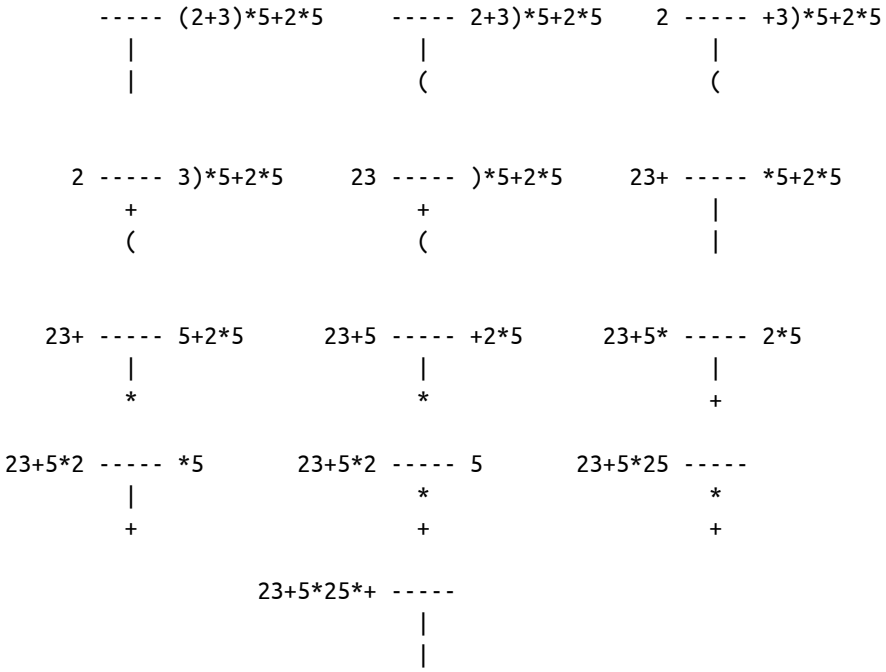
Израчунавање вредности израза је опет комбинација превођења у постфиксни облик и израчунавања вредности постфиксног израза.

Проблем се решава слично као код потпуно заграђених израза, али овај пут се мора обраћати пажња на приоритет и асоцијативност оператора. Решење се може направити рекурзивним спустом, али се тиме нећемо бавити у овом курсу. Кључна дилема је шта радити у ситуацији када се прочита `op2` у изразу облика `i1 op1 i2 op2 i3` где су `i1`, `i2` и `i3` три израза (било броја било израза у заградама), а `op1` и `op2` два оператора. У том тренутку на излазу ће се налазити израз `i1` преведен у постфиксни облик и иза њега израз `i2` преведен у постфиксни облик, док ће се оператор `op1` налазити на врху стека оператора. Уколико `op1` има већи приоритет од оператора `op2` или уколико им је приоритет исти, али је асоцијативност лева, тада је потребно прво

израчунавати израз `i1 op1 i2` тиме што се оператор `op1` са врха стека пребаци на излаз. У супротном (ако `op2` има већи приоритет или ако је приоритет исти, а асоцијативност десна) оператор `op1` остаје на стеку и изнад њега се поставља оператор `op2`.

Ово је један од многих алгоритама које је извео Едсгер Дејкстра и назива се на енглеском језику *Shunting yard algortihm*, што би се могло слободно превести као алгоритам сортирања железничких вагона. Замислимо да израз треба да пређе са једног на други крај пруге. На прузи се налази споредни колосек (пруга је у облику слова T и споредни колосек је усправна црта). Делови израза прелазе са десног на леви крај (замислимо да иду по горњој ивици слова T). Бројеви увек прелазе директно. Оператори се увек задржавају на споредном колосеку, али тако да се пре него што оператор уђе на споредни колосек са њега на излаз пребацују сви оператори који су вишег приоритета у односу на текући или имају исти приоритет као текући а лево су асоцијативни. И отворене заграде се постављају на споредни колосек, а када наиђе затворена заграда са споредног колосека се уклањају сви оператори до отворене заграде. Када се исцрпи цео израз на десној страни, сви оператори са споредног колосека се пребацују на леву страну. Јасно је да споредни колосек има понашање стека, тако да имплементацију можемо направити коришћењем стека на који ћемо стављати операторе.

Илуструјмо ову железничку аналогију једним примером.




```
#include <iostream>
#include <string>
#include <stack>

using namespace std;

// provera da li je karakter aritmeticki operator
bool jeOperator(char c) {
  return c == '+' || c == '-' || c == '*' || c == '/';
}

// prioritet datog operatora
int prioritet(char c) {
  if (c == '+' || c == '-')
    return 1;
  else if (c == '*' || c == '/')
    return 2;
  // greska
  return 0;
}
```

```
// primenjuje datu operaciju na dve vrednosti na vrhu steka,
// zamenjujući ih sa rezultatom primene te operacije
// vracamo informaciju o tome da li operator uspesno primenjen ili je
// doslo do deljenja nulom
bool primeni(stack<char>& operatori, stack<int>& vrednosti) {
  // operator se nalazi na vrhu steka operatora
  char op = operatori.top(); operatori.pop();
  // operandi se nalaze na vrhu steka operatora
  int op2 = vrednosti.top(); vrednosti.pop();
  int op1 = vrednosti.top(); vrednosti.pop();

  // izracunavamo vrednost izraza
  int v = 0;
  if (op == '+') v = op1 + op2;
  else if (op == '-') v = op1 - op2;
  else if (op == '*') v = op1 * op2;
  else if (op == '/') {
    // deljenje nulom
    if (op2 == 0)
      return false;
    v = op1 / op2;
  }
  // postavljamo ga na stek operatora
  vrednosti.push(v);
  // operator je uspesno primenjen
  return true;
}


// izracunavamo vrednost izraza
// vracamo informaciju o tome da li je vrednost uspesno izracunata ili
// je doslo do deljenja nulom
bool vrednost(const string& izraz, int& v) {
  stack<int> vrednosti;
  stack<char> operatori;

  // analiziramo sve karaktere u ulaznom izrazu
  int i = 0;
  while (i < izraz.length()) {
    if (isdigit(izraz[i])) {
      // brojevne konstante postavljamo na stek
      v = izraz[i] - '0';
      i++;
      while (i < izraz.length() && isdigit(izraz[i]))
        v = 10 * v + (izraz[i++] - '0');
      vrednosti.push(v);
    } else if (izraz[i] == '(') {
      // otvorene zagrade postavljamo na stek
      operatori.push('(');
      i++;
    } else if (izraz[i] == ')') {
      // izračunavamo vrednost izraza u zagradi
      while (operatori.top() != '(')
        if (!primeni(operatori, vrednosti))
          return false;
      // uklanjamo otvorenu zagradu
      operatori.pop();
```

```
      i++;
    } else if (jeOperator(izraz[i])) {
       // obrađujemo sve prethodne operatore višeg prioriteta
       while (!operatori.empty() && jeOperator(operatori.top()) &&
              prioritet(operatori.top()) >= prioritet(izraz[i]))
         if (!primeni(operatori, vrednosti))
           return false;
       // stavljamo operator na stek
       operatori.push(izraz[i]);
       i++;
    }
  }

  // izračunavamo sve preostale operacije
  while (!operatori.empty())
    if (!primeni(operatori, vrednosti))
      return false;

  // vrednost izraza se nalazi na vrhu steka
  v = vrednosti.top();
  return true;
}

int main() {
  // citamo linije do kraja ulaza
  string s;
  while (cin >> s) {
    // pokusavamo da izracunamo vrednost izraza
    int rez;
    if (vrednost(s, rez))
      cout << rez << endl;
    else
      cout << "deljenje nulom" << endl;
  }
}
```

Задатак: Вредност мин-макс израза

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Вредност потпуно заграђеног мин-макс израза

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Фреквенцијски стек

Програм извршава две врсте операција са структуром података која донекле подсећа на стек.

- Операција `0 x` поставља елемент `x` на врх стека.
- Операција `1` уклања елемент који се најчешће појављује од свих елемената који су тренутно на стеку. Ако је више таквих елемената, уклања се онај који је последњи додат. Нпр. ако су на стеку елементи `1 2 2 1 3`, уклања се елемент `1` и стек постаје `1 2 2 3`.

Улаз: Са стандардног улаза се учитава број операција n ($1 \leq n \leq 10^5$), а затим n операција (свака у посебном реду).

Излаз: Опис излазних података.

Пример

Улаз	*Излаз*
12 0 1 0 2 0 2 0 1 0 3 0 2 1 1 1 1 1 1	2 1 2 3 2 1

Решење

Основна идеја решења је да се уместо јединственог стека елементи чувају на више стекова организованих по фреквенцијама тј. броју појављивања елемента. Прво појављивање неког елемента стављаћемо на стек првих појављивања, друго појављивање на стек других појављивања итд. Да бисмо знали на који стек треба ставити наредно појављивање неког елемента, помоћу асоцијативног низа (речника, мапе) одржаваћемо број појављивања сваког елемента. Одржаваћемо и максималну фреквенцију. Ако се додавањем неког елемента повећава максимална фреквенција, додајемо нови стек. Приликом уклањања елемента, уклањаћемо врх последњег стека, тј. стека који одговара тој највећој фреквенцији. Ако се након тога тај стек испразни, смањујемо максималну фреквенцију.

Прикажимо рад алгоритма на примеру додавања бројева `1 2 2 1 3 2`.

- Прво се додаје број `1` на стек елемената који се појављују само једном. Максимална фреквенција постаје 1.

  ```
  1: 1
  ```

- Након тога се додаје број `2` на стек елемената који се појављују само једном.

  ```
  1: 1, 2
  ```

- Затим се додаје број `2` на стек елемената који се појављују два пута. Максимална фреквенција постаје 2.

  ```
  1: 1, 2
  2: 2
  ```

- Затим се додаје број `1` на стек елемената који се појављују два пута.

  ```
  1: 1, 2
  2: 2, 1
  ```

- Затим се додаје број `3` на стек елемената који се појављују једном.

  ```
  1: 1, 2, 3
  2: 2, 1
  ```

- На крају се додаје број `2` на стек елемената који се појављују три пута. Максимална фреквенција постаје 3.

  ```
  1: 1, 2, 3
  2: 2, 1
  3: 2
  ```

Прикажимо сада уклањање елемената док се стек не испразни.

- Пошто је максимална фреквенција 3, уклања се врх стека елемената са фреквенцијом 3 и исписује се број 2. Пошто се тај стек испразнио максимална фреквенција се смањује на 2.

  ```
  1: 1, 2, 3
  2: 2, 1
  ```

- Пошто је максимална фреквенција 2, уклања се врх стека елемената са фреквенцијом 2 и исписује се број 1.

  ```
  1: 1, 2, 3
  2: 2
  ```

- Пошто је максимална фреквенција 2, уклања се врх стека елемената са фреквенцијом 2 и исписује се број 2. Пошто се тај стек испразнио максимална фреквенција се смањује на 1.

  ```
  1: 1, 2, 3
  ```

- Након тога се редом скидају и исписује елементи 3, 2 и на крају 1 (тада се максимална фреквенција смањује на нулу).

```cpp
#include <iostream>
#include <stack>
#include <map>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  // frekvencija tj. broj pojavljivanja svakog elementa
  map<int, int> frekv;
  // niz stekova - na steku broj i čuvaju se i-ta pojavljivanja svih
  // elemenata u originalnom steku
  map<int, stack<int>> stekovi;
  // najveca frekvencija nekog elementa u originalnom steku
  int maksFrekv = 0;

  // obradjujemo n naredbi
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int naredba;
    cin >> naredba;
    if (naredba == 0) {
      // ucitavamo element
      int x;
      cin >> x;
      // uvecavamo njegov broj pojavljivanja
      frekv[x]++;
      // u skladu sa tim ga postavljamo na njemu odgovarajuci stek
      stekovi[frekv[x]].push(x);
      // azuriramo najvecu frekvenciju nekog elementa
      maksFrekv = max(maksFrekv, frekv[x]);
    } else if (naredba == 1) {
      // uklanjamo element sa vrha steka elemenata koji se pojavljuju
      // najcesce
      int x = stekovi[maksFrekv].top();
      stekovi[maksFrekv].pop();
      cout << x << '\n';
      // smanjujemo njegov broj pojavljivanja
      frekv[x]--;
      // azuriramo najvecu frekvenciju nekog elementa
      if (stekovi[maksFrekv].empty())
        maksFrekv--;
    }
  }

  return 0;
```

```
}
```

Задатак: Стек и максимуми

Дечак слаже тањире један на други. Тањири могу бити различитог полупречника. У сваком кораку он може да дода нови тањир на врх гомиле, да скине тањир са врха (ако је гомила непразна) и да постави питање који је полупречник највећег тањира на гомили.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим n упита:

- упит облика `0 r` означава да се на врх гомиле додаје нови тањир полупречника `r` (`r` је природан број мањи од 10^9).
- упит облика `1` означава да се са врха гомиле скида тањир (овај упит се никада неће поставити ако је гомила празна).
- упит облика `2` означава захтев да се на стандардни излаз испише највећи полупречник тањира који је тренутно на гомили (овај упит се никада неће поставити ако је гомила празна).

Излаз: На стандардни излаз исписати резултате свих упита `2`.

Пример

Улаз	*Излаз*
`10`	`3`
`0 1`	`4`
`0 3`	`1`
`0 2`	
`2`	
`1`	
`0 4`	
`2`	
`1`	
`1`	
`2`	

Објашњење

На почетку се додају тањири полупречника 1, 3 и 2 и гомила тањира тада има садржај `1 3 2`. Максимум је тада 3. Након тога се скида тањир са врха па гомила постаје `1 3`. Додаје се тањир полупречника 4, па гомила постаје `1 3 4`. Максимум је тада 4. Након тога се скидају тањири полупречника 3 и 4, па на гомили остаје само тањир полупречника 1 (гомила тада има садржај `1`). Максимум је тада 1.

9.2 Ред

Задатак: Сегмент највећег просека

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Приметимо да је проблем налажења сегмента дужине k чији је просек највећи, еквивалентан проблему налажења сегмента дужине k највећег збира. Просек сегмента добијамо дељењем суме сегмента са дужином сегмента, која је у овом задатку константа и износи k, тако да је просек највећи када је збир највећи.

Смањивање употребљене меморије коришћењем реда

Претходно решење је временски ефикасно, али се користи превише меморије. Наиме, није неопходно у меморији истовремено чувати све елементе низа, већ је довољно чувати само елементе текућег сегмента (ово може бити битно у случајевима када је меморија критичан ресурс и када је k знатно мање од n). Приликом учитавања сваког новог елемента почетни елемент тренутног сегмента се уклања, а последњи елемент се додаје на крај сегмента. Ово указује на то да је за чување текућег сегмента погодна структура података ред, која нам омогућава да додајемо елементе на крај и уклањамо елементе са почетка (први елемент који је додат први бива и уклоњен из реда). У језику C++ можемо употребити колекцију `queue`. Елементе у ред додајемо

методом `push`, уклањамо их методом `pop` (она не враћа вредност). Елемент на почетку реда се може очитати методом `front`.

```
#include <iostream>
#include <queue>
#include <iterator>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // broj elemenata niza i duzina segmenta
  int n, k;
  cin >> k >> n;

  // red u kojem u svakom trenutku cuvamo tekuci segment
  queue<double> q;

  // ucitavamo prvi segment duzine k, smestamo elemente u red i
  // racunamo mu sumu
  double suma = 0.0;
  for (int i = 0; i < k; i++) {
    double x; cin >> x;
    q.push(x);
    suma += x;
  }

  // trenutna maksimalna suma segmenta i indeks njenog pocetka
  int maxPocetak = 0;
  double maxSuma = suma;

  for (int i = 1; i <= n-k; i++) {
    // ucitavamo naredni element
    double x; cin >> x;
    // azuriramo sumu
    suma = suma - q.front() + x;
    // menjamo "najstariji" element u redu
    q.pop(); q.push(x);
    // ako je potrebno, azuriramo maksimum
    if (suma >= maxSuma) {
      maxSuma = suma;
      maxPocetak = i;
    }
  }

  // ispisujemo pocetak poslednjeg segmenta sa maksimalnom sumom
  // (ujedno i prosekom)
  cout << maxPocetak << endl;

  return 0;
}
```

Задатак: Последњих k линија

Напиши програм који исписује k последњих линија учитаних са стандардног улаза.

Улаз: Са стандардног улаза се учитава број k ($1 \leq k \leq 100$), а затим једна по једна линија текста (њих највише 10^6).

Излаз: На стандардни излаз исписати последњих k линија (претпоставити да је увек учитано барем k линија).

Пример

Улаз	*Излаз*
2 ispisati poslednjih k linija	poslednjih k linija

Задатак: Сортирање - сви испред мањи или сви испред већи

Бројеви у низу су такви да за сваки елемент важи или да су сви елементи испред њега мањи од њега или да су сви елементи испред њега већи од њега. Нпр. низ $5, 8, 12, 4, 2, 13, 19, 1$ задовољава то својство. Напиши програм који у линеарној сложености сортира тај низ.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим n елемената низа (елементи су дати у једној линији, раздвојени размацима).

Излаз: На стандардни излаз исписати сортиране елементе низа (раздвојене размаком).

Пример

Улаз	*Излаз*
8 5 8 12 4 2 13 19 1	1 2 4 5 8 12 13 19

Решење

Ако се не узме у обзир специфичност улазних података, низ се може сортирати било којим алгоритмом за сортирање низа бројева (слично као у задатку Сортирање бројева).

Ако се користи библиотечка функција сортирања, временска сложеност овог алгоритма је $O(n \log n)$, док је меморијска сложеност $O(n)$.

```
#include <iostream>
#include <deque>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  deque<int> a;

  int x; cin >> x;
  a.push_back(x);
  for (int i = 0; i < n; i++) {
    cin >> x;
    if (a.back() < x)
      a.push_back(x);
    else
      a.push_front(x);
  }

  for (int x : a)
    cout << x << " ";
  cout << endl;

  return 0;
}
```

Решимо задатак индуктивном конструкцијом. Претпоставимо да обрађујемо један по један по један елемент и да смо већ сортирали префикс елемената испред текућег. Ако је он већи од свих елемената испред себе у

полазном низу, тада га треба додати на крај сортираног префикса, а ако је мањи од свих елемената испред себе, треба га додати на почетак сортираног префикса.

Прикажимо сортирање низа `5, 8, 12, 4, 2, 13, 19, 1`.

- На почетку је ред празан, па у њега додајемо 5 (ред постаје `5`).
- 8 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `5, 8`).
- 12 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `5, 8, 12`).
- 4 је мање од последњег елемента у реду, па га додајемо на почетак (ред постаје `4, 5, 8, 12`).
- 2 је мање од последњег елемента у реду, па га додајемо на почетак (ред постаје `2, 4, 5, 8, 12`).
- 13 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `2, 4, 5, 8, 12, 13`).
- 19 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `2, 4, 5, 8, 12, 13, 19`).
- 1 је мање од последњег елемента у реду, па га додајемо на почетак (ред постаје `1, 2, 4, 5, 8, 12, 13, 19`).

За ефикасну имплементацију потребно је користити структуру података која допушта ефикасно додавање елемената на почетак и на крај и то може бити ред са два краја или двоструко повезана листа. У језику C++ можемо употребити `deque` или `list`. У језику C# можемо употребити `LinkedList`. Да би се проверило да ли је елемент већи од свих испред себе или мањи од свих испред себе, довољно је упоредити га са било којим од тих елемената (на пример, са последњим елементом у реду). Када је ред празан, први елемент можемо убацити било на почетак, било на крај. Да бисмо избегли проверу да ли је ред празан приликом обраде сваког елемента, први елемент можемо убацити у празан ред пре него што почнемо са обрадом осталих елемената.

И меморијска и временска сложеност овог алгоритма је $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  sort(begin(a), end(a));
  for (int i = 0; i < n; i++)
    cout << a[i] << " ";
  cout << endl;

  return 0;
}
```

Задатак: Јосифов проблем

Ђаци седе у кругу обележени бројевима од 0 до $n-1$ и играју се разбрајалице тако да у сваком бројању један ђак испадне. Бројање креће од ђака 0 и сваки m-ти ђак испада. Напиши програм који одређује који ђак ће остатати последњи.

Улаз: У првој линији стандардног улаза налази се почетни број ђака n ($1 \leq n \leq 10^5$), а у другом дужина бројалице m ($2 \leq m \leq n$).

Излаз: На стандардни излаз исписати број преосталог ђака.

Пример

Улаз	*Излаз*
`8`	`6`
`3`	

Објашњење

Ђаци који седе у кругу на почетку и након сваког испадања су:

```
0 1 2 3 4 5 6 7
0 1 3 4 5 6 7
0 1 3 4 6 7
1 3 4 6 7
1 3 6 7
3 6 7
3 6
6
```

Решење

Кружну листу можемо једноставно реализовати коришћењем реда. У сваком кораку бројања једног ђака са почетка реда ћемо пребацити на крај реда. Након пребацивања $m - 1$ ученика, оног који је на почетку реда трајно избацујемо.

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
  int n;
  cin >> n;
  int m;
  cin >> m;
  queue<int> red;
  for (int i = 0; i < n; i++)
    red.push(i);
  while (red.size() > 1) {
    for (int i = 0; i < m - 1; i++) {
      red.push(red.front());
      red.pop();
    }
    red.pop();
  }
  cout << red.front() << endl;
  return 0;
}
```

Задатак: Прости чиниоци 235

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Коришћење редова

У сваки од три низа елементи се додају на крај, а узимају се са почетка. Зато је уместо низова могуће користити редове са два краја. За имплементацију можемо користити `deque` у језику C++.

Број елемената у низовима полако расте током извршавања алгоритма, али се неки елементи укањају са почетка тих редова, па је употреба меморије мања него када се елементи чувају у векторима (иако вероватно не и асимптотски).

```
#include <iostream>
#include <queue>
#include <algorithm>

using namespace std;
```

```
int main() {
  int n;
  cin >> n;
  deque<long long> a2, a3, a5;
  a2.push_back(2);
  a3.push_back(3);
  a5.push_back(5);
  long long t = 1;
  for (int i = 0; i < n; i++) {
    t = min({a2.front(), a3.front(), a5.front()});
    a2.push_back(2*t);
    a3.push_back(3*t);
    a5.push_back(5*t);
    while (a2.front() == t) a2.pop_front();
    while (a3.front() == t) a3.pop_front();
    while (a5.front() == t) a5.pop_front();
  }
  cout << t << endl;
  return 0;
}
```

Задатак: Најдужи сегмент који садржи узастопне бројеве

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Максимална бијекција

Филмски продуцент организује вечеру на коју жели да позове глумце. Да би се глумци осећали пријатно на вечери, продуцент жели да обезбеди да је сваки глумац присутан на вечери омиљен глумац неког другог глумца присутног на вечери. Сваки од n глумаца, потенцијалних гостију, одабрао је свог омиљеног глумца из тог скупа глумаца (при чему није искључено и да је неки глумац одабрао сам себе). Напиши програм који одређује највећи подскуп тог скупа глумаца који садржи глумце које продуцент може позвати на вечеру.

Улаз: Са стандардног улаза уноси се број n ($1 \leq n \leq 50000$) који представља број глумаца који су гласали, а затим и редом редни бројеви омиљеног глумца сваког глумца (сви бројеви су између 0 и $n-1$).

Излаз: На стандардни излаз испиши највећи број глумаца који могу присуствовати вечери.

Пример

Улаз	*Излаз*
7	3
2	
0	
0	
4	
4	
3	
5	

Објашњење

На вечеру могу бити позвани глумци са бројевима $0, 2, 4$. Глумац 0 је омиљени глумац глумца 2, глумац 2 је омиљени глумац глумца 0, док је глумац 4 сам себи омиљен.

Решење

Гласови глумаца одређују функцију f дефинисану на скупу $\{0, 1, \ldots, n-1\}$. Нека је скуп S скуп глумаца који су позвани на вечеру. Да би сваки глумац био омиљен глумац неком другом глумцу из скупа S, потребно је да рестрикција функције f на скуп S буде "на" (тј. да за сваку слику постоји оригинал који се слика у ту слику). Пошто је скуп S коначан, на основу Дирихлеовог принципа, она ће уједно бити и "1-1" (за сваку слику ће постојати тачно један оригинал који се у њу слика). Заиста, ако би неки глумац на вечери био омиљен за

два различита глумца, неком глумцу на вечери би недостајао глумац коме би он био омиљен. Функија која је истовремено “на” и “1-1” зове се бијекција.

Провера свих подскупова

Директан, али веома неефикасан начин да се овај задатак реши је да се наброје сви могући подскупови и да се за сваки од њих провери да ли је рестрикција f на тај подскуп бијекција. Генерисање свих подскупова можемо урадити по узору на задатак Сви подскупови, а проверу да ли је f бијекција можемо урадити пребројавањем оригинала који се сликају у сваку од слика - ако је функција бијекција тада се у сваку слику слика тачно један оригинал. Пребројавање можемо урадити коришћењем асоцијативног низа и то или низа бројача или, још боље, низа индикатора (истинитосних вредности) које указују на то да ли је за дату слику већ пронађен оригинал, слично као у задатку Различите цифре).

Још боље решење је да генеришемо само оне подскупове за које унапред знамо да је функција на њима бијекција. Рекурзивној функцији за генерисање подксупова прослеђујемо текући елемент i, скуп оригинала мањих од i који су укључени у подскуп који се гради и скуп њихових слика (оба скупа можемо представити низом битова). Елементи мањи од i који нису у скупу оригинала изостављени су из подскупа који се гради, док је статус елемената већих или једнаких од i непознат (они потенцијално могу бити и укључени и искључени из скупа оригинала). Инваријанта која се одржава је да се никоја два оригинала не сликају ни у једну слику, да скуп слика тачно представља слике свих оригинала, као и да су све слике досадашњих оригинала које су мање од i укључене у скуп оригинала (оне слике које су веће или једнаке i биће укључене накнадно, ако то не нарушава инјективност).

Излаз из рекурзије је случај $i = n$. Тада знамо да су све слике скупа оригинала укључене у скуп оригинала, па се та два скупа поклапају, и функција је бијекција на том скупу (јер знамо да је инјективна). Функција тада може да врати кардиналност текућег скупа оригинала.

У сваком позиву рекурзивне функције проверавамо две могућности:

- Тренутни елемент i може да се укључи у скуп оригинала. Ово је могуће само ако је слика елемента i није избачена из скупа (што се дешава ако је $f(i) < i$, а $f(i)$ се не налази у низу оригинала), јер би се у супротном нарушио услов да је свака слика која је мања од i укључена у низ оригинала. Такође, слика не сме бити већ укључена у низ слика (тј. у низу оригинала не сме да се налази већ неки други елемент који има исту слику као i), јер би се тада додавањем елемента i нарушила инјективност.
- Друга могућност је да се i изостави из скупа оригинала. То је могуће само ако i није слика неког мањег елемента (ако јесте, онда i мора бити укључен у скуп оригинала).

Пошто подскупова има 2^n, овај алгоритам је недопустиво неефиксан (чак и у варијанти у којој се неки скупови прескачу током генерисања).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// do sada popunjeni skup originala (strogo manjih od i) i skup
// njihovih slika prosirujemo elementima vecim ili jednakim od i
// invarijanta je da je funkcija injektivna na skupu originala i
// da je slika svakog originala ukljucena u skup slika i da su slike
// svih elemenata manjih od i ukljucene u skup originala
int bijekcija(const vector<size_t>& f,
              vector<bool>& originali, vector<bool>& slike, size_t i) {
  // skup originala se ne moze vise prosiriti
  // na osnovu invarijante funkcija je bijekcija na skupu originala
  if (i == f.size())
    return count(begin(originali), end(originali), true);

  int rez = 0;
  // ako f[i] nije ranije oznacena kao izbacena iz skupa originala
  // i ako f[i] nije vec slika nekog elementa iz niza originala
  if ((f[i] >= i || originali[f[i]]) && !slike[f[i]]) {
```

```
    // ubacujemo i u skup originala i f[i] u skup slika
    originali[i] = true;
    slike[f[i]] = true;
    // prosirujemo vecim elementima i azuriramo rezultat
    int rez_sa_i = bijekcija(f, originali, slike, i+1);
    rez = rez_sa_i;
    // izbacujemo i iz skupa originala i f[i] iz skupa slika
    // da bismo se vratili u pocetnu poziciju
    originali[i] = false;
    slike[f[i]] = false;
  }
  // ako i nije slika nekog elementa, onda on moze biti preskocen
  if (!slike[i]) {
    // prosirujemo elementima vecim od i (i je preskocen)
    int rez_bez_i = bijekcija(f, originali, slike, i+1);
    // azuriramo rezultat ako je to potrebno
    if (rez_bez_i > rez)
      rez = rez_bez_i;
  }
  return rez;
}

int bijekcija(const vector<size_t>& f) {
  int n = f.size();
  vector<bool> originali(n, false), slike(n, false);
  return bijekcija(f, originali, slike, 0);
}

int main() {
  int n;
  cin >> n;
  vector<size_t> f(n);
  for (int i = 0; i < n; i++)
    cin >> f[i];

  cout << bijekcija(f) << endl;
  return 0;
}
```

Елиминација елемената

Ефикасан алгоритам можемо направити ако пронађемо потребан услов да елемент буде део скупа S. Наиме, сваки елемент скупа S мора бити слика тачно једног елемента скупа S. Ако је сваки елемент скупа X (домена функције f) слика тачно једног елемента скупа X, тада је f бијекција на скупу X. У супротном мора да постоји елемент који није слика ни једног елемента скупа X и тај елемент не може бити део скупа S. Када тај елемент уклонимо (заједно са његовом сликом), добијамо скуп који је за један елемент мањи и на који можемо применити исти поступак (суштински, имамо описан индуктивни тј. рекурзивни поступак).

Овај приступ се може једноставно имплементирати тако што за сваки елемент скупа X израчунамо број елемената који се сликају у њега (то можемо урадити коришћењем једноставног, асоцијативног низа, слично као у задатку Фреквенције речи). Можемо одржавати радну листу (ред) елемената у које се не слика ни један елемент (након израчунавања броја елемената који се сликају у сваки од елемената скупа X, све бројеве за које је вредност у асоцијативном низу нула, убацујемо у радну листу). Након тога, све док се радна листа не испразни, узимамо један по један елемент из радне листе, избацујемо га из скупа X и зато смањујемо број елемената који се сликају у слику тог избаченог елемента (умањујемо вредност у асоцијативном низу). Ако се установи да се након тога вредност слике у асоцијативном низу смањила на нулу, тада се слика убацује у радну листу.

```
#include <iostream>
#include <vector>
```

```cpp
#include <queue>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> f(n);
  for (int i = 0; i < n; i++)
    cin >> f[i];
  vector<int> ulazniStepen(n, 0);
  for (int i = 0; i < n; i++)
    ulazniStepen[f[i]]++;
  queue<int> q;
  for (int i = 0; i < n; i++)
    if (ulazniStepen[i] == 0)
      q.push(i);
  int broj_elemenata = n;
  while (!q.empty()) {
    int i = q.front(); q.pop();
    broj_elemenata--;
    if (--ulazniStepen[f[i]] == 0)
      q.push(f[i]);
  }
  cout << broj_elemenata << endl;
  return 0;
}
```

9.3 Ред са приоритетом

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање уз помоћ реда са приоритетом

Сортирање бројева се може извршити коришћењем реда са приоритетом. Користи се алгоритам *сортирања уз помоћ хипа* (енгл. *heap sort*) који је варијација алгоритма сортирања селекцијом (енгл. *selection sort*) у којем се, подсетимо се, у сваком кораку најмањи елемент доводи на почетак низа. Алгоритам хип сорт користи чињеницу да је одређивање и уклањање најмањег елемента из реда са приоритетом прилично ефикасна операција. Стога се сортирање може реализовати тако што се сви елементи уметну у ред са приоритетом, из кога се затим проналази и уклања један по један најмањи елемент.

И убацивање елемената у ред са приоритетом и избацивање елемената из реда са приоритетом обично је сложености $O(\log k)$, где је k број елемената у реду са приоритетом). Стога је укупна сложеност овог алгоритма сортирања $O(n \log n)$.

```cpp
// ovo je način da se u C++-u definiše red sa prioritetom u kome su
// elementi poređani u opadajućem redosledu prioriteta (ovde, vrednosti)
priority_queue<int, vector<int>, greater<int>> Q;
// učitavamo sve elemente niza i ubacujemo ih u red
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int ai;
  cin >> ai;
```

```
  Q.push(ai);
}
// vadimo jedan po jedan element iz reda i ispisujemo ga
while (!Q.empty()) {
  cout << Q.top() << endl;
  Q.pop();
}
```

Задатак: Збир k најбољих

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Збирови квадрата

Поређајмо све парове природних бројева (a, b), где је $0 \leq a \leq b$ редом, у односу на вредност њихових квадрата (ако два пара имају исту вредност збира квадрата, онда их ређамо редом, у односу на вредност првог броја a). Та серија елемената почиње овако:

$$\begin{aligned} 0^2 + 0^2 &= 0 \\ 0^2 + 1^2 &= 1 \\ 1^2 + 1^2 &= 2 \\ 0^2 + 2^2 &= 4 \\ 1^2 + 2^2 &= 5 \\ 2^2 + 2^2 &= 8 \\ &\dots \end{aligned}$$

Неки елементи серије се понављају (на пример, $0^2 + 5^2 = 25$, $3^2 + 4^2 = 25$).

Напиши програм који одређује k-ти члан те серије (бројање почиње од 0).

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^4$), такав да се разматрају само парови (a, b) за које је $0 \leq a \leq b \leq n$, а затим из наредног реда број k (он је већи или једнак од 0, а строго мањи од броја елемената листе).

Излаз: На стандардни излаз исписати елемент описане серије збирова квадрата на позицији k (позиције се броје од нуле) – исписати бројеве a, b и $a^2 + b^2$, раздвојене једним размаком.

Пример 1

Улаз	*Излаз*
2	2 2 8
5	

Пример 2

Улаз	*Излаз*
10	1 5 26
15	

Решење

Груба сила - сортирање низа

Решење грубом силом подразумева да угнежђеним петљама формирамо низ свих тројки $(a^2 + b^2, a, b)$ и да га сортирамо на основу вредности збира квадрата.

Пошто парова има $O(n^2)$, меморијска сложеност је $O(n^2)$, а временска $O(n^2 \log n^2) = O(n^2 \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <tuple>
#include <functional>

using namespace std;
```

```
long long zbir_kvadrata(int a, int b) {
  return (long long)a * a + (long long)b * b;
}

int main() {
  int n;
  cin >> n;
  int k;
  cin >> k;

  // sve uredjene trojke stavljamo u niz
  vector<tuple<long long, int, int>> v;
  for (int b = 0; b <= n; b++)
    for (int a = 0; a <= b; a++)
      v.emplace_back(zbir_kvadrata(a, b), a, b);

  // sortiramo niz - trojke se porede leksikografski, prvo po prvoj
  // komponenti (zbiru kubova), a ako je zbir kubova jednak, onda po
  // prvom sabirku
  sort(begin(v), end(v));

  // ispisujemo element na poziciji k
  int a, b;
  long long s;
  tie(s, a, b) = v[k];
  cout << a << " " << b << " " << s << endl;

  return 0;
}
```

Обједињавање сортираних серија

Меморијски ефикасније решење можемо добити ако проблем сведемо на проблем обједињавања неколико сортираних низова бројева (слично као у задатку). Размотримо следеће низове бројева (задате по врстама):

$$\begin{array}{llll} (0,0) & & & \\ (0,1) & (1,1) & & \\ (0,2) & (1,2) & (2,2) & \\ \multicolumn{4}{c}{\ldots} \\ (0,n) & (1,n) & (2,n) & \ldots \quad (n,n) \end{array}$$

Сви ови низови су такви да су сортирани по вредности збира квадрата (наиме, знамо да је је $a^2 + b^2 < (a+1)^2 + b^2$), да су међусобно дисјунктни и да закључно са последњим они покривају све парове бројева $0 \leq a \leq b \leq n$. Тражена сортирана листа свих тих парова и њихових збирова квадрата се зато добија обједињавањем датих $n+1$ сортираних листа (свака је одређена неком вредношћу b од 0 до n), тако што у сваком тренутку узимамо најмањи елемент са почетка неке од њих. Пошто постоји прилично велики број листа ($O(n)$), најмањи од њих ћемо најефикасније наћи ако почетне елементе чувамо у реду са приоритетом (уређеном по вредности збира квадрата). На почетку у ред стављамо почетни елемент сваке листе (то су бројеви од $(0,b)$, за $0 \leq b \leq n$). У сваком кораку узимамо најмањи елемент из реда (a,b) и ако је $a < b$ у ред додајемо елемент $(a+1,b)$ (он је наредни елемент листе одређене другим елементом b).

Рецимо и да смо овај принцип могли искористити и да није било дато горње ограничење n. Једини проблем је то што не бисмо знали које елементе на почетку треба да убацимо у ред са приоритетом. Стога бисмо почетне елементе листа могли да додајемо "лењо" тј. да почетак наредне листе $(0,b)$ додамо у ред у тренутку када је ред празан или када је елемент на врху реда такав да му је збир квадрата већи или једнак b^2 (наравно, наредну вредност b бисмо одржавали у посебној променљивој).

У реду са приоритетом се у сваком тренутку чува највише по један елемент сваке од листа које се обједињавају. Пошто има $O(n)$ листа, меморијска сложеност овог решења је $O(n)$. Пошто дужина резултујуће листе и

вредност k припадају класи $O(n^2)$, ажурирање реда са приоритетом се врши $O(n^2)$ пута. Свако ажурирање је сложености $O(\log n)$ (јер се у њему налази највише $O(n)$ елемената), па је временска сложеност $O(n^2 \log n)$.

```
#include <iostream>
#include <tuple>
#include <queue>
#include <functional>

using namespace std;

long long zbir_kubova(int a, int b) {
  return (long long)a * a * a + (long long)b * b * b;
}

int main() {
  int n;
  cin >> n;
  int k;
  cin >> k;

  // red neopadajuce sortiran na osnovu vrednosti zbira kubova (prve
  // komponente uredjene trojke), a zatim na osnovu manjeg sabirka
  // (druge komponente uredjene trojke)
  priority_queue<tuple<long long, int, int>,
                 vector<tuple<long long, int, int>>,
                 greater<tuple<long long, int, int>>> pq;

  // ubacujemo pocetni element svake liste u red sa prioritetom
  for (int b = 0; b <= n; b++)
    pq.emplace(zbir_kubova(0, b), 0, b);

  // odredjujemo element objedinjene liste na poziciji k
  for (int K = 0; K < k; K++) {
    // skidamo element sa pocetka reda
    int a, b;
    long long s;
    tie(s, a, b) = pq.top();
    pq.pop();

    // dodajemo u red naredni element liste u kojoj se nalazi skinuti element
    if (a < b)
      pq.emplace(zbir_kubova(a + 1, b), a+1, b);
  }

  // element objedinjene liste na poziciji k je onaj koji je trenutno
  // na pocetku reda
  int a, b;
  long long s;
  tie(s, a, b) = pq.top();
  cout << a << " " << b << " " << s << endl;

  return 0;
}
```

Сви елементи (a, b), за $0 \leq a \leq b \leq n$ се могу разложити у дисјунктне сортиране листе (задате по врстама):

$$
\begin{array}{cccccc}
(0,0) & (0,1) & (0,2) & \dots & (0,n-1) & (0,n) \\
 & (1,1) & (1,2) & \dots & (1,n-1) & (1,n) \\
 & & & \dots & & \\
 & & & & (n-1,n-1) & (n-1,n) \\
 & & & & & (n,n)
\end{array}
$$

Свака од ових $n+1$ листа одређена је једном вредношћу $0 \leq a \leq n$ и почиње елементом (a,a). Пошто је $a^2+b^2 < a^2+(b+1)^2$, свака од њих је сортирана по вредности збира квадрата. За њихово обједињавање могуће је такође употребити ред са приоритетом. Скидамо елемент (a,b) са почетка реда и ако је $b < n$ у ред убацујемо $(a,b+1)$.

И у овом случају се задатак може решити без унапред задатог горњег ограничења n, једино што тада не бисмо у старту у ред убацили почетке свих листа (a,a) за $0 \leq a \leq n$, већ бисмо једну по једну листу додавали по потреби. Нови елемент (a,a) можемо додати када је елемент на почетку реда мањи или једнак од a^2+b^2 или када је ред празан (може се показати да се ово друго неће никада догодити, јер ако не постоји горње ограничење број елемената у реду са приоритетом ће с временом само расти и увек ће одговарати броју листа чије је обједињавање започето).

Меморијска сложеност овог решења је $O(n)$, а временска $O(n^2 \log n)$.

```
#include <iostream>
#include <tuple>
#include <queue>
#include <functional>

using namespace std;

long long zbir_kvadrata(int a, int b) {
  return (long long)a * a + (long long)b * b;
}

int main() {
  int n;
  cin >> n;
  int k;
  cin >> k;

  // red neopadajuce sortiran na osnovu vrednosti zbira kvadrata (prve
  // komponente uredjene trojke), a zatim na osnovu manjeg sabirka
  // (druge komponente uredjene trojke)
  priority_queue<tuple<long long, int, int>,
                 vector<tuple<long long, int, int>>,
                 greater<tuple<long long, int, int>>> pq;

  // ubacujemo pocetni element svake liste u red sa prioritetom
  for (int a = 0; a <= n; a++)
    pq.emplace(zbir_kvadrata(a, a), a, a);

  // odredjujemo element objedinjene liste na poziciji k
  for (int K = 0; K < k; K++) {
    // skidamo element sa pocetka reda
    int a, b;
    long long s;
    tie(s, a, b) = pq.top();
    pq.pop();

    // dodajemo u red naredni element liste u kojoj se nalazi skinuti element
    if (b < n)
      pq.emplace(zbir_kvadrata(a, b+1), a, b+1);
```

```
  }

  // element objedinjene liste na poziciji k je onaj koji je trenutno
  // na pocetku reda
  int a, b;
  long long s;
  tie(s, a, b) = pq.top();
  cout << a << " " << b << " " << s << endl;

  return 0;
}
```

Задатак: Степени природних бројева

Размотримо серију бројева који су степени природних бројева (за основу и експонент строго веће од 1), уређену растући. Њен почетни део је $a_0 = 4 = 2^2$, $a_1 = 8 = 2^3$, $a_2 = 9 = 3^2$, $a_3 = 16 = 4^2 = 2^4$, $a_4 = 25 = 5^2$, $a_5 = 27 = 3^3$, $a_6 = 32 = 2^5$ итд. Напиши програм који одређује члан a_k.

Улаз: Са стандардног улаза се учитава број k ($1 \leq k \leq 10^6$).

Излаз: На стандардни излаз исписати члан a_k.

Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
6	32	1000000	979850535876

Решење

Једно решење може бити засновано на томе да се за сваки природни број редом испита да ли је степен неког природног броја. То може бити реализовано растављањем броја на просте чиниоце $x = p_1^{\alpha_1} \dots p_m^{\alpha_m}$ (то можемо урадити као у задатку Растављање на просте чиниоце). Број x је степен неког природног броја (за експонент строго већи од 1) ако и само ако је број N који је НЗД бројева $\alpha_1, \dots, \alpha_m$ строго већи од 1 (важи да је $x = (p_1^{\frac{\alpha_1}{N}} \dots p_m^{\frac{\alpha_m}{N}})^N$).

Факторизација сваког елемента x је сложености $O(\sqrt{M})$, где је M највећи прост фактор броја x. Експериментално се веома грубо може проценити да је k-ти елемент серије степена природних бројева реда величине $O(k^2)$. Стога се сложеност веома грубо може проценити на $O(k^2\sqrt{k})$. Меморијска сложеност овог решења је константна.

```
#include <iostream>

using namespace std;

typedef long long ll;

// NZD brojeva a i b
int nzd(int a, int b) {
  if (b == 0)
    return a;
  return nzd(b, a % b);
}

// provera da li je x stepen nekog manjeg prirodnog broja
bool stepen(ll x) {
  // rastavljamo x na proste cinioce p1^a1 * ... * pm^am

  // nzd brojeva a1, ..., am (0 je neutral za nzd)
  int n = 0;

  // tekuci faktor
  ll d = 2;
  while (d * d <= x) {
```

```
    // broj pojavljivanja faktora d u broju x
    int a = 0;
    while (x % d == 0) {
      x /= d;
      a++;
    }
    // prosirujemo faktorizaciju faktorom d^a, pa azuriramo i
    // nzd stepena dosadasnjih faktora
    n = nzd(n, a);
    // ako je nzd vec sada 1, ostace jedan do kraja
    if (n == 1)
      return false;
    d++;
  }
  // preostao je poslednji prost faktor, njegov stepen je 1
  if (x > 1)
    return false;
  return true;
}

int main() {
  int k;
  cin >> k;
  // kandidat kojeg proveravamo
  ll x = 2;
  // broj odredjenih elemenata serije
  int i = 0;
  while(true) {
    // uvecavamo x dok ne nadjemo neki stepen prirodnog broja
    while (!stepen(x))
      x++;
    // ako je ovo trazeni clan serije, prekidamo petlju
    if (i == k)
      break;
    // trazimo naredni clan serije, krenuvsi od x+1
    i++; x++;
  }
  cout << x << endl;
  return 0;
}
```

Уређена серија степена природних бројева може се добити обједињавањем сортираних серија које одговарају појединачним експонентима. То су серије:

$$2^2, 3^2, 4^2, 5^2, 6^2, \ldots$$

$$2^3, 3^3, 4^3, 5^3, 6^3, \ldots$$

$$2^4, 3^4, 4^4, 5^4, 6^4, \ldots$$

итд.

Рецимо и да је довољно посматрати само серије са простим експонентима (јер се елементи серија са сложеним експонентима већ јављају у ранијим серијама).

Пошто број серија који се обједињују полако расте, користићемо обједињавање уз помоћ реда са приоритетом. У њему чувамо текући елемент сваке серије. У сваком кораку вадимо најмањи елемент из реда са

приоритетом и у ред додајемо наредни елемент серије којој припада извађени елемент (ако је извађен x^e, додајемо $(x+1)^e$). Пошто се неки елементи серије могу представити на више начина као степени природних бројева (на пример, $a_3 = 16 = 2^4 = 4^2$), у сваком кораку ћемо из реда са приоритетом са почетка вадити сва појављивања тог степена (и за сваки извађени елемент, додавати наредни елемент одговарајуће листе).

Пошто не знамо унапред које ће листе бити потребне, кренућемо само од прве и мало по мало ћемо укључивати нове листе. Када је почетак наредне листе (број облика 2^e) мањи или једнак од елемента на почетку реда, убацивaћемо 2^e у ред и тако укључити листу која њиме почиње у процес обједињавања.

До k-тог елемента се стиже у $O(k)$ корака (корака је мало више од k, јер постоје дупликати тј. елементи који се изражавају као степени природних бројева на више начина, али њих је јако мало). С обзиром на ограничења дата у тексту задатка у реду може бити највише четрдесетак листи (јер је 2^{40} приближно једнако милионитом члану серије). Стога се избацивање и убацивање у ред може проценити као операција константне сложености (логаритам броја 40 је веома мали број), па се укупна сложеност може проценити са $O(k)$.

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <tuple>

using namespace std;

typedef long long ll;
typedef tuple<ll, int, int> trojka;

ll stepen(ll x, int s) {
  if (s == 0)
    return 1;
  else if (s % 2 == 0)
    return stepen(x*x, s / 2);
  else
    return x * stepen(x, s - 1);
}

int main() {
  int k;
  cin >> k;
  priority_queue<trojka, vector<trojka>, greater<trojka>> pq;

  // svaka lista pocinje elementom 2^eksp

  // krecemo objedinavanje od liste 2^2, 3^2, 4^2, ...
  int eksp = 2;
  ll sledeci = stepen(2, eksp);
  pq.emplace(sledeci, 2, eksp);

  // sledeca lista je 2^3, 3^3, 4^3 - nju cemo ukljuciti kasnije
  eksp++;
  sledeci = stepen(2, eksp);

  for (int i = 0; i < k; i++) {
    // pamtimo vrednost stepena na pocetku reda
    ll xs = get<0>(pq.top());
    // vadimo iz reda sva pojavljivanja te vrednosti stepena
    while (xs == get<0>(pq.top())) {
      // za svaku skinutu vrednost x^s ubacujemo u red (x+1)^s
      int x, s;
      tie(xs, x, s) = pq.top();
      pq.pop();
```

```
      pq.emplace(stepen(x+1, s), x+1, s);
    }

    // proveravamo da li je potrebno da u objedinjavanje ukljucimo i
    // novu listu koja pocinje elementom 2^eksp
    if (sledeci <= get<0>(pq.top())) {
      pq.emplace(sledeci, 2, eksp);
      eksp++;
      sledeci = stepen(2, eksp);
    }
  }

  // k-ti element serije se nalazi na pocetku reda
  cout << get<0>(pq.top()) << endl;

  return 0;
}
```

Задатак: K-ти највећи збир пара елемената два низа

Дата су два низа која садрже природне бројеве. Напиши програм који одређује k-ти највећи збир који се може добити када се сабере један елемент првог и један елемент другог низа.

Улаз: Са стандардног улаза се учитава број m ($1 \leq m \leq 5000$), а затим из наредног реда m елемената првог низа раздвојених размаком. Из наредног реда се учитава број n ($1 \leq n \leq 5000$), а затим из наредног реда n елемената другог низа раздвојених размаком. Елементи оба низа су природни бројеви између 0 и 10^6. На крају се учитава број k ($0 \leq k < mn$).

Излаз: На стандардни излаз збир који се налази на позицији k у низу који би се добио када би се низ свих збирова парова једног елемента првог и једног елемента другог низа сортирао нерастуће (позиције се броје од нуле).

Пример 1

Улаз
```
3
1 5 3
3
6 4 2
4
```

Излаз
```
7
```

Објашњење

Збирови који се могу добити, поређани од највећег ка најмањем су $5+6=11, 5+4=9, 3+6=9, 5+2=7, 3+4=7, 1+6=7, 3+2=5, 1+4=5, 1+2=3$, па се на позицији 4 налази збир 7.

Пример 2

Улаз
```
5
5 3 8 6 1
6
1 10 9 7 12 2
9
```

Излаз
```
15
```

Решење

Груба сила

Решење грубом силом подразумева да се формира низ свих збирова, да се сортира нерастуће и да се прочита елемент са позиције k.

Парова елемената има $m \cdot n$, па је меморијска сложеност квадратна и износи $O(mn)$. Временском сложеношћу доминира сортирање и она износи $O(mn \log (mn))$.

```
#include <iostream>
#include <vector>
#include <tuple>
#include <queue>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
  int m;
  cin >> m;
  vector<int> a(m);
  for (int i = 0; i < m; i++)
    cin >> a[i];

  int n;
  cin >> n;
  vector<int> b(n);
  for (int i = 0; i < n; i++)
    cin >> b[i];

  int k;
  cin >> k;

  // formiramo sve zbirove parova elemenata dva niza
  vector<int> zbirovi;
  zbirovi.reserve(m*n);
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      zbirovi.push_back(a[i] + b[j]);

  // sortiramo niz zbirova nerastuce
  sort(begin(zbirovi), end(zbirovi), greater<int>());

  // ispisujemo k-ti element sortranog niza zbirova
  cout << zbirovi[k] << endl;

  return 0;
}
```

Обједињавање сортираних низова

Проблем се може свести на проблем обједињавања неколико сортираних низова, који се не морају истовремено чувати у меморији. Наиме, ако сортирамо оба низа опадајуће, можемо разматрати следеће низове:

$$a_0 + b_0, a_0 + b_1, \ldots, a_0 + b_{n-1},$$

$$a_1 + b_0, a_1 + b_1, \ldots, a_1 + b_{n-1},$$

...

$$a_{m-1}+b_0, a_{m-1}+b_1, \ldots, a_{m-1}+b_{n-1}.$$

Сви они су сортирани нерастуће и могу се објединити коришћењем реда са приоритетом. У почетку у ред убацујемо први елемент сваке листе тј. све збирове облика $a_i + b_0$. У сваком кораку избацујемо највећи збир из реда и додајемо у ред наредни елемент листе којој он припада. Да бисмо након вађења елемента из реда могли додати наредни елемент листе којој он припада, поред вредности збира $a_i + a_j$ (на основу којих је ред уређен) у реду морамо чувати и индексе i и j. Заправо довољно је чувати само индекс j, јер се на основу збира $z = a_i + b_j$, збир $a_i + b_{j+1}$ може добити као $z - b_j + b_{j+1}$.

У реду се у сваком тренутку налази највише m елемената. Пошто се чувају и оригинални низови (да би се сортирали), меморијска сложеност је $O(m + n)$. Ажурирање реда врши се k пута. Пошто је сложеност иницијалних сортирања низова \$, сложеност једног ажурирања реда је $O(\log m)$, а важи $k < mn$, временска сложеност је $O(m \log m + n \log n + mn \log m)$. Сложеношћу јасно доминира фаза обједињавања, па се временска сложеност може проценити и само са $O(mn \log m)$.

```
#include <iostream>
#include <vector>
#include <tuple>
#include <queue>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
  int m;
  cin >> m;
  vector<int> a(m);
  for (int i = 0; i < m; i++)
    cin >> a[i];

  int n;
  cin >> n;
  vector<int> b(n);
  for (int i = 0; i < n; i++)
    cin >> b[i];

  int k;
  cin >> k;

  // sortiramo nizove opadajuce
  sort(begin(a), end(a), greater<int>());
  sort(begin(b), end(b), greater<int>());

  // objedinjavamo sortirane nizove
  // a[i] + b[0], ..., a[i] + b[n-1], za svako i od 0 do m-1
  // u redu cuvamo zbirove a[i] + b[j] i pozicije j
  priority_queue<pair<int, int>> pq;

  // dodajemo u red prvi element svakog niza
  for (int i = 0; i < m; i++)
    pq.emplace(a[i] + b[0], 0);

  // k puta azuriramo red
  for (int K = 0; K < k; K++) {
```

```
    // skidamo element sa vrha reda
    int j, z;
    tie(z, j) = pq.top(); pq.pop();
    // ako lista u kojoj je bio skinuti elemnt nije ispraznjena,
    // u red dodajemo njen naredni element
    if (j + 1 < n)
      pq.emplace(z - b[j] + b[j+1], j+1);
  }

  // element na poziciji k je trenutno na pocetku reda
  cout << pq.top().first << endl;

  return 0;
}
```

Задатак: Ажурирање медијане

У заводу за статистику желе да што непристрасније процене која је просечна плата. Закључили су да израчунавање аритметичке средине може дати мало искривљену слику јер неколико људи са веома високим платама могу значајно повећати просек. Зато су одлучили да уместо аритметичке средине израчунају медијану која се добија тако што се све плате поређају у неопадајући низ и онда се узме средишњи елемент тог низа. Ако у низу има паран број елемената, онда се за медијану узима аритметичка средина два средишња елемента. На пример, ако медијана низа бројева 1, 2, 4, 7, 9 је 4 (јер је он средишњи), а низа бројева 1, 2, 4, 5, 7, 9 је 4.5 (јер је то аритметичка средина бројева 4 и 5 који су средишњи елементи). Подаци о платама пристижу у завод, а софтвер мора да може да у сваком тренутку да податак о медијани до тада унетих плата.

Улаз: Са стандардног улаза се уносе линије, све до краја стандардног улаза. Линија или садржи слово `d` и затим износ плате раздвојен размаком (цео број), што значи да се уноси податак о новој плати или садржи слово `m` што значи да је потребно на стандардни излаз исписати податак о медијани до тада унетих плата. Прва линија сигурно садржи `d`.

Излаз: На стандардном излазу су исписане тражене медијане, свака у посебном реду, заокружене на једну децималу.

Пример

Улаз	*Излаз*
`d 5`	`6.0`
`d 7`	`6.5`
`d 6`	
`m`	
`d 8`	
`m`	

Решење

Израчунавање медијане сваки пут

Директан начин да се задатак реши је тај да се све учитане плате смештају у низ (или још боље вектор тј. листу, пошто не знамо унапред колико ће плата бити учитано) и да се сваки пут када се затражи израчунавање медијане позове функција која рачуна медијану низа (слично као што је демонстрирано у задатку Медијана).

Та функција може бити заснована на сортирању и читању средишњег елемента (тј. средишњих елемената када је низ парне дужине).

Сложеност таквог решења зависи од сложености сортирања који је $O(n \log(n))$ када се користи бибилотечка функција сортирања, тако да је укупна сложеност $O(n^2 \cdot \log(n))$, при чему претпостављамо да ће се медијана рачунати $O(n)$ пута за низ који има $O(n)$ елемената (тј. да су и број додавања и број упита за медијаном одређени бројем n).

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
```

```
using namespace std;

double medijana(vector<int>& a) {
  int n = a.size();
  sort(begin(a), end(a));
  if (n % 2 == 1)
    return a[n/2];
  else
    return (a[n/2 - 1] + a[n/2]) / 2.0;
}


int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  vector<int> a;
  while (true) {
    string s;
    if (!(cin >> s))
      break;
    if (s == "m")
      cout << fixed << showpoint << setprecision(1)
           << medijana(a) << '\n';
    else if (s == "d") {
      int x;
      cin >> x;
      a.push_back(x);
    }
  }
  return 0;
}
```

Медијана се може наћи и без сортирања целог низа, коришћењем идеје партиционисања тј. алгоритма брзе селекције (QuickSelect).

У језику C++ ово је најлакше остварити помоћу функције `nth_element`. Рецимо и да је у случају парне димензије функцију `nth_element` потребно позвати два пута, да би се одредила два средишња елемента.

Сложеност проналажења медијане алгоритмом брзе селекције је у најгорем случају $O(n)$, што доводи до укупног решења сложености $O(n^2)$.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;

double medijana(vector<int>& a) {
  int n = a.size();
  if (n % 2 == 1) {
    nth_element(begin(a), next(begin(a), n / 2), end(a));
    int m = a[n/2];
    return m;
  } else {
    nth_element(begin(a), next(begin(a), n / 2 - 1), end(a));
    int m1 = a[n/2 - 1];
    nth_element(begin(a), next(begin(a), n / 2), end(a));
    int m2 = a[n/2];
    return (m1 + m2) / 2.0;
  }
}
```

```
int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  vector<int> a;
  while (true) {
    string s;
    if (!(cin >> s))
      break;
    if (s == "m")
      cout << fixed << showpoint << setprecision(1)
           << medijana(a) << '\n';
    else if (s == "d") {
      int x;
      cin >> x;
      a.push_back(x);
    }
  }
  return 0;
}
```

Одржавање сортираног низа

Боља решења се могу добити коришћењем боље организованих структура података. Један начин је да се наметне инваријанта да се у сваком тренутку плате чувају сортирано. Приликом додавања нове плате, она се може додати на крај, а затим уметнути на своје место (слично као у алгоритму сортирања уметањем InsertionSort, приказаном у задатку Сортирање бројева).

Сложеност корака уметања је $O(n)$ (иако би се позиција на коју се умеће могла наћи бинарном претрагом, само уметање захтева померање свих чланова иза те позиције, што тражи линеарно време). Само проналажење медијане онда захтева константно време (читање средишњег тј. два средишња елемента). Ако се умеће n елемената, укупна сложеност је $O(n^2)$.

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

double medijana(vector<int>& a) {
  int n = a.size();
  if (n % 2 == 1)
    return a[n/2];
  else
    return (a[n/2 - 1] + a[n/2]) / 2.0;
}

void umetni_poslednji(vector<int>& a) {
  int n = a.size();
  int tmp = a[n-1];
  int i;
  for (i = n-2; i >= 0 && a[i] > tmp; i--)
    a[i+1] = a[i];
  a[i+1] = tmp;
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  vector<int> a;
  while (true) {
    string s;
```

```
    if (!(cin >> s))
      break;
    if (s == "m")
      cout << fixed << showpoint << setprecision(1)
           << medijana(a) << '\n';
    else if (s == "d") {
      int x;
      cin >> x;
      a.push_back(x);
      umetni_poslednji(a);
    }
  }
  return 0;
}
```

Балансирано стабло (мултискуп)

Подаци се могу чувати у оквиру балансираног бинарног стабла, и у сваком тренутку се чува итератор који указује на средишњи елемент и да се приликом сваког наредног додатка он ажурира (помера се за једно место улево, тј. удесно). Ова техника је детаљно објашњена у задатку Сумњиве трансакције и приказано је како се она лако може имплементирати коришћем типа `multiset`.

```
#include <iostream>
#include <iomanip>
#include <set>
#include <string>
using namespace std;

multiset<int> a;
multiset<int>::const_iterator sredina;

double medijana() {
  if (a.size() % 2 != 0)
    return *sredina;
  else
    return (*prev(sredina) + *sredina) / 2.0;
}

void dodaj(int x) {
  if (a.size() == 0) {
    a.insert(x); sredina = begin(a);
  } else {
    a.insert(x);
    if (x < *sredina) {
      if (a.size() % 2 != 0)
        advance(sredina, -1);
    } else {
      if (a.size() % 2 == 0)
        advance(sredina, 1);
    }
  }
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  while (true) {
    string s;
    if (!(cin >> s))
      break;
```

```
    if (s == "m")
      cout << fixed << showpoint << setprecision(1)
           << medijana() << '\n';
    else if (s == "d") {
      int x;
      cin >> x;
      dodaj(x);
    }
  }
  return 0;
}
```

Два хипа

Још један веома добар начин је да у сваком тренутку у једној (рећи ћемо левој) колекцији чувамо све елементе који су мањи од средишњег, а у другој (рећи ћемо десној) све оне који су већи или једнаки средишњем (ако постоји паран број елемената, те две колекције треба да садрже исти број елемената, а ако постоји непаран број елемената, десна колекција може да садржи један елемент више). Ако има непаран број елемената, тада је медијана једнака најмањем елементу десне колекције, а у супротном је једнака аритметичкој средини између највећег елемента леве и најмањег елемента десне колекције. Сваки нови елемент се пореди са најмањим елементом десне колекције и ако је мањи или једнак њему убацује се у леву колекцију, а ако је већи од њега, убацује се у десну колекцију. Тада се проверава да ли се средина променила. Ако се десило да лева колекција има више елемената од десне (што не допуштамо), највећи елемент леве колекције треба да пребацимо у десну. Ако се десило да у десној колекцији има два елемента више него у левој, тада најмањи елемент десне колекције пребацујемо у леву. Дакле, лева колекција треба да буде таква да лако можемо да пронађемо и избацимо њен највећи елемент, а десна да буде таква да лако можемо да пронађемо и избацимо њен најмањи елемент, при чему обе колекције морају да подрже ефикасно убацивање произвољних елемената. Јасно је да те колекције треба да буду хипови (у језику C++ можемо употребити `priority_queue`) у којима се најмања тј. највећа вредност може очитати у константном времену, уклонити у логаритамском, исто колико је потребно и да се уметне нови елемент.

```
#include <iostream>
#include <iomanip>
#include <queue>
#include <string>
#include <vector>
#include <functional>
using namespace std;

priority_queue<int, vector<int>, greater<int>> veci_od_sredine;
priority_queue<int, vector<int>, less<int>> manji_od_sredine;

double medijana() {
  if (manji_od_sredine.size() == veci_od_sredine.size())
    return (manji_od_sredine.top() + veci_od_sredine.top()) / 2.0;
  else
    return veci_od_sredine.top();
}

void dodaj(int x) {
  if (veci_od_sredine.empty())
    veci_od_sredine.push(x);
  else {
    if (x <= veci_od_sredine.top())
      manji_od_sredine.push(x);
    else
      veci_od_sredine.push(x);
    if (manji_od_sredine.size() > veci_od_sredine.size()) {
      veci_od_sredine.push(manji_od_sredine.top());
```

```
      manji_od_sredine.pop();
    } else if (veci_od_sredine.size() > manji_od_sredine.size() + 1) {
      manji_od_sredine.push(veci_od_sredine.top());
      veci_od_sredine.pop();
    }
  }
}


int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  while (true) {
    string s;
    if (!(cin >> s))
      break;
    if (s == "m")
      cout << fixed << showpoint << setprecision(1)
           << medijana() << '\n';
    else if (s == "d") {
      int x;
      cin >> x;
      dodaj(x);
    }
  }
  return 0;
}
```

Мало спорија имплементација је помоћу мултискупа у којем се и очитавање и уклањање минимума тј. максимума, као и уметање новог елемента могу извршити у логаритамском времену (пошто је у језику C++ `multiset` сортиран мултискуп, најмањи елемент се може добити као први, а највећи као последњи) .

```
#include <iostream>
#include <iomanip>
#include <set>
#include <string>

using namespace std;

// multiskup elemenata koji su manji od medijane
multiset<int> manji_od_sredine;

// multiskup elemenata koji su veci od medijane
multiset<int> veci_od_sredine;

int najveci(const multiset<int>& s) {
  return *prev(end(s));
}

int najmanji(const multiset<int>& s) {
  return *begin(s);
}

void ukloni_najveci(multiset<int>& s) {
  s.erase(prev(end(s)));
}

void ukloni_najmanji(multiset<int>& s) {
  s.erase(begin(s));
```

```
}

double medijana() {
  if (manji_od_sredine.size() == veci_od_sredine.size())
    return (najveci(manji_od_sredine) + najmanji(veci_od_sredine)) / 2.0;
  else
    return najmanji(veci_od_sredine);
}

void dodaj(int x) {
  if (veci_od_sredine.empty())
    veci_od_sredine.insert(x);
  else {
    if (x <= najmanji(veci_od_sredine))
      manji_od_sredine.insert(x);
    else
      veci_od_sredine.insert(x);
    if (manji_od_sredine.size() > veci_od_sredine.size()) {
      veci_od_sredine.insert(najveci(manji_od_sredine));
      ukloni_najveci(manji_od_sredine);
    } else if (veci_od_sredine.size() > manji_od_sredine.size() + 1) {
      manji_od_sredine.insert(najmanji(veci_od_sredine));
      ukloni_najmanji(veci_od_sredine);
    }
  }
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  while (true) {
    string s;
    if (!(cin >> s))
      break;
    if (s == "m")
      cout << fixed << showpoint << setprecision(1)
           << medijana() << '\n';
    else if (s == "d") {
      int x;
      cin >> x;
      dodaj(x);
    }
  }
  return 0;
}
```

Задатак: Сумњиве трансакције

Банка жели да упозори купце на сумњиве активности на њиховом рачуну. Приликом сваке трансакције посматра се средишња вредност m (медијана) претходних d трансакција и ако је текућа трансакција већа или једнака од двоструке вредности m, издаје се упозорење. Средишња вредност низа може да се израчуна тако што се пронађе средишњи елемент у низу сортираном по величини (ако је број елемената паран, онда се узима аритметичка средина између два елемента око средине низа). Напиши програм који на основу списка трансакција одређује број упозорења која ће банка се послати.

Улаз: Са стандардног улаза се уноси цео број n ($5 \leq n \leq 10^5$) који одређује укупан број трансакција, затим број d ($1 \leq d \leq n$), а затим n вредности трансакција (природни бројеви између 1 и 200). Сви бројеви се налазе у посебним линијама.

Излаз: На стандардни излаз исписати један природан број који представља број сумњивих трансакција.

Пример

Улаз	*Излаз*
8 3 2 5 3 4 3 6 2 9	2

Задатак: Силуета града

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Једно ефикасније решење можемо постићи ако зграде обрађујемо слева надесно. Силуета се мења само у тачкама у којма почиње или се завршава нека зграда. Стога можемо направити низ карактеристичних тачака који садржи све почетке и крајеве зграда и информацију о томе да ли је текућа тачка почетак или крај и обрађивати те тачке у неопадајућем редоследу (потребно је низ тих карактеристичних тачака сортирати). Код сваке карактеристичне тачке потребно је да одредимо висину силуете након те тачке. Ту висину ћемо одредити као највећу висину свих зграда које почињу лево од те карактеристичне тачке (укључујући евентуално и ту тачку) а чији се десни крај не налази лево од те тачке (укључујући евентуално и њу). За то су нам потребне ефикасне структуре података.

Потребно је да одржавамо структуру података у коју ћемо убацивати зграду по зграду како наилазе (када се наиђе на њихов леви крај), избацивати зграде како пролазе (када се наиђе на њихов десни крај) и у сваком тренутку моћи ефикасно да пронађемо максимум тренутно убачених зграда. Проблем са овим захтевима је то што је тешко ефикасно их остварити истовремено. Наиме, ако бисмо чували податке о зградама некако уређене на основу њихових висина, ефикасно бисмо проналазили највишу, али би избацивање зграда на основу позиција крајева било компликовано (јер зграде не би биле уређене по том критеријуму). Са друге стране, ако бисмо зграде чували некако уређене по координатама крајева, проналажење оне са максималном висином би било неефикасно. Први приступ је ипак бољи, јер не можемо никако да се одрекнемо могућности ефикасног налажења максимума. Структура која нам то омогућава је ред са приоритетом (хип) који је уређен на основу висина зграда. Проблем са таквим редом је то што је избацивање зграде када се наиђе на њен десни крај проблематично (ред са приоритетом нам даје веома ефикасно избацивање зграде која је највиша, али не и осталих зграда). Кључни трик, који се веома често може употребити код коришћења редова са приоритетом је да избацивање елемената из реда одложимо и да у структури података допустимо чување података који су по неком критеријуму застарели и који не би више требало да се употребљавају. Наиме, претпоставимо да се у реду са приоритетом налазе све зграде на чији смо почетак до сада наишли. Када желимо да пронађемо највишу зграду од њих која се још није завршила, можемо да размотримо зграду на врху реда. Могуће је да се она још није завршила и у том случају она представља решење. У супротном, ако се та зграда завршила, њој није више место у реду и можемо да је избацимо из реда. Међутим, за разлику од тренутка када смо наишли на њен десни крај, када је њено избацивање било компликовано, у овом тренутку се она налази на врху реда и избацивање се може извршити веома ефикасно. Дакле, анализираћемо и избацивaћемо једну по једну највишу зграду са врха реда све док не дођемо до зграде која се још није завршила.

Остаје још питање како нову промену интегрисати у постојећу силуету. То није тешко, али је потребно обратити пажњу на неколико специјалних случајева (који углавном наступају услед тога што више зграда могу имати исти почетак или крај). Инваријанта коју желимо да наметнемо на силуету је да су x-координате свих узастопних промена различите и да не постоје две узастопне промене са истом висином (друга промена је тада вишак). Ако последња промена у силуети има исту x-координату као и промена која се убацује, онда се уместо додавања нове промене ажурира висина те последње промене, ако је то потребно (ако је нова висина већа од постојеће). Тиме се може десити да након ажурирања последња и претпоследња промена имају исту висину, па је у том случају потребно уклонити последњу промену. На крају, ако нова промена има исту висину као и последња промена у силуети, нема потребе да се додаје. Овим се инваријанта одржава.

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <queue>
using namespace std;

struct zgrada {
  int a, b, h;
  zgrada(int a = 0, int b = 0, int h = 0)
    : a(a), b(b), h(h) {
  }
};

struct promena {
  int x, h;
  promena(int x = 0, int h = 0)
    : x(x), h(h) {
  }
};

// integrisemo promenu (x, h) u tekucu siluetu
void dodajPromenu(vector<promena>& silueta, int x, int h) {
  // broj promena u silueti
  int n = silueta.size();
  // ako silueta nije prazna
  if (n > 0) {
    // (xb, hb) je poslednja promena u silueti
    int xb = silueta[n-1].x;
    int hb = silueta[n-1].h;

    // ako je (x, h) na istoj x koordinati kao poslednja promena
    if (xb == x) {
      // ako je visina h veca od poslednje, samo azuriramo visinu
      if (h > hb) {
        silueta[n-1].h = h;
        // ako postoji pretposlednja promena i ako je ona na istoj
        // visini onda se poslednja promena moze izbaciti
        if (n > 1 && silueta[n - 2].h == h)
          silueta.pop_back();
      }
    } else if (hb != h)
      // (x, h) je potpuno razlicita od poslednje, pa je samo dodajemo
      silueta.push_back(promena(x, h));
  } else
    // silueta je prazna pa samo dodajemo (x, h)
    silueta.push_back(promena(x, h));
}

vector<promena> napraviSiluetu(vector<zgrada>& zgrade) {
  vector<promena> silueta;

  // sortiramo sve zgrade na osnovu pocetka
  struct PorediPocetak {
    bool operator() (const zgrada& z1, const zgrada& z2) {
      return z1.a < z2.a;
    }
  };
```

```
  sort(begin(zgrade), end(zgrade), PorediPocetak());

  // pravimo vektor svih pocetnih i krajnjih tacaka zgrada i za svaku
  // tacku belezimo da li je pocetna
  vector<pair<int, bool>> tacke(2 * zgrade.size());
  int i = 0;
  for (auto z : zgrade) {
    tacke[i++] = make_pair(z.a, true);
    tacke[i++] = make_pair(z.b, false);
  }
  // sortiramo sve znacajne tacke na osnovu koordinate
  sort(begin(tacke), end(tacke), [](auto p1, auto p2){
      return p1.first < p2.first;
    });


  // cuvamo visine do sada obradjenih zgrada tako da veoma brzo mozemo
  // da odredimo zgradu najvece visine - koristimo red sa prioritetom
  struct PorediVisinu {
    bool operator() (const zgrada& z1, const zgrada& z2) {
      return z1.h < z2.h;
    }
  };
  priority_queue<zgrada, vector<zgrada>, PorediVisinu> pq;

  // obilazimo sve znacajne tacke sleva nadesno
  int z = 0;
  for (auto p : tacke) {
    int x = p.first;
    bool je_pocetak = p.second;

    // dodajemo u red sve zgrade koje pocinju na trenutnoj koodrinati
    if (je_pocetak)
      while (z < zgrade.size() && zgrade[z].a == x)
        pq.push(zgrade[z++]);

    // trazimo najvisu visinu do sada zapocete zgrade
    // eliminisemo zgrade koje su do sada zavrsene
    while (!pq.empty() && pq.top().b <= x)
      pq.pop();
    int h = !pq.empty() ? pq.top().h : 0;

    // integrisemo visinu najvise zgrade na trenutnoj koordinati u
    // tekucu siluetu
    dodajPromenu(silueta, x, h);
  }

  // vracamo rezultat
  return silueta;
}

int main() {
  // ucitavamo podatke o zgradama
  int n;
  cin >> n;
  vector<zgrada> zgrade(n);
  for (int i = 0; i < n; i++) {
    int a, b, h;
```

```
    cin >> a >> b >> h;
    zgrade[i] = zgrada(a, b, h);
  }

  // gradimo siluetu
  vector<promena> s = napraviSiluetu(zgrade);

  // ispisujemo rezultat
  for (auto p : s)
    cout << p.x << " " << p.h << endl;

  return 0;
}
```

Задатак: Најкраћи сегмент збира бар K

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Збирови префикса уз одсецање

Алгоритам заснован на префиксним збировима се може убрзати одсецањем. Наиме, ако за неко i и j важи да је $z_{i+1} - z_j \geq K$, тада се z_j може елиминисати из даље анализе. Наиме, ако за неко $i < i' < n$ важи да је $z_{i'+1} - z_j \geq K$, тако добијени сегмент $[j, i']$ ће сигурно бити дужи од сегмента $[j, i]$, па неће моћи да буде најкраћи. Зато у сваком кораку из низа префикса избацујемо све елементе за које је $z_{i+1} - z_j \geq K$ и не поредимо их са наредним збировима $z_{i'+1}$. Пошто се елементи избацују из низа, губимо информацију о њиховој позицији (за вредност z_j више не знамо индекс j), па стога морамо уместо вредности збирова z_j у низу чувати парове (z_j, j).

Иако одсецање може уштедети одређени број провера тј. одређени број извршавања унутрашње петље, сложеност најгорег случаја остаје $O(n^2)$, јер у неким примерима до одсецања уопште не долази (на пример, када су збирови свих сегмената мањи од K).

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  int k;
  cin >> k;
  int n;
  cin >> n;
  // duzina najkraceg segmenata (+beskonacno na pocetku)
  int minBrojElemenata = n + 1;
  // niz svih preostalih zbirova prefiksa - uz svaki zbir zj prvih j
  // elemenata niza pamti se i vrednost j
  vector<pair<int, int>> zbirovi;
  // zbir prefiksa [0, i]
  int zbir = 0;
  // zbir prvih 0 elemenata niza je 0
  zbirovi.emplace_back(zbir, 0);
  // analiziramo sve leve krajeve segmenta
  for (int i = 0; i < n; i++) {
    // ucitavamo element x = a[i] i uvecavamo zbir prefiksa
    int x; cin >> x;
    zbir += x;
```

```
    // analiziramo sve segmente [j, i] za jos neeliminisane prefikse j
    // za sve one ciji je zbir bar k azuriramo minimalnu duzinu i
    // eliminisemo ih, dok sve ostale zadrzavamo (prebacujuci ih na
    // pocetak niza)
    int m = 0;
    for (auto z : zbirovi)
      if (zbir - z.first >= k) {
        int brojElemenata = i - z.second + 1;
        minBrojElemenata = min(minBrojElemenata, brojElemenata);
      } else
        zbirovi[m++] = z;
    // skracujemo niz ostavljajuci samo zbirove prebacene na pocetak
    zbirovi.erase(next(zbirovi.begin(), m), zbirovi.end());

    // izracunati zbir prvih i+1 clanova niza ubacujemo u niz zbirova
    zbirovi.emplace_back(zbir, i+1);
  }

  // ispisujemo rezultat
  if (minBrojElemenata <= n)
    cout << minBrojElemenata << endl;
  else
    cout << "-" << endl;

  return 0;
}
```

Ред са приоритетом

Унутрашња петља пролази кроз све збирове z_j за $j < i$, тражећи оне најмање, за које важи да је $z_{i+1} - z_j \geq K$ и избацујући их из даљег разматрања. Време би се могло уштедити ако би се збирови префикса уместо у низу чували у структури података која би омогућавала да брзо пронађемо најмањи елемент и да га избацимо, а то је управо ред са приоритетом. Дакле, у сваком кораку проверавамо да ли за најмањи елемент у реду са приоритетом z_j важи да је $z_{i+1} - z_j \geq K$ и ако јесте избацујемо тај елемент из реда (ажурирајући при том дужину минималног сегмента, ако је то потребно). То се понавља све док се ред не испразни или док се не наиђе на најмањи елемент реда z_j за који важи да је $z_{i+1} - z_j < K$. Тада у ред убацујемо z_{i+1} и прелазимо на наредни корак спољне петље.

Пошто се убацивање и избацивање елемената у ред са приоритетом у коме има m елемената врши у сложености $O(\log m)$, а очитавање најмањег елемента у сложености $O(1)$, укупна сложеност овог приступа је $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>

using namespace std;

int main() {
  int k;
  cin >> k;
  int n;
  cin >> n;
  // duzina najkraceg segmenata (+beskonacno na pocetku)
  int minBrojElemenata = n + 1;

  // red sa prioritetom u kom se cuvaju zbirovi prefiksa
  // za svaki zbir zj prvih j elemenata niza cuva se i broj j
  // red je uredjen neopadajuce tj. na vrhu se nalaze najmanji zbirovi
```

```
  priority_queue<pair<int, int>, vector<pair<int, int>>,
                 greater<pair<int, int>>> zbirovi;
  // zbir prefiksa [0, i]
  int zbir = 0;
  // zbir prvih 0 elemenata je 0
  zbirovi.emplace(zbir, 0);
  // analiziramo sve leve krajeve segmenta
  for (int i = 0; i < n; i++) {
    // ucitavamo element x = a[i] i uvecavamo zbir prefiksa
    int x; cin >> x;
    zbir += x;

    // analiziramo segmente [j, i] za dovoljno male vrednosti zbira zj
    while (!zbirovi.empty() && zbir - zbirovi.top().first >= k) {
      // azuriramo duzinu najkraceg segmenta
      int brojElemenata = i - zbirovi.top().second + 1;
      minBrojElemenata = min(minBrojElemenata, brojElemenata);
      // izbacujemo najmanji zbir iz reda
      zbirovi.pop();
    }

    // izracunati zbir prvih i+1 clanova niza ubacujemo u red zbirova
    zbirovi.emplace(zbir, i+1);
  }

  // ispisujemo rezultat
  if (minBrojElemenata <= n)
    cout << minBrojElemenata << endl;
  else
    cout << "-" << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Својство 132

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Највећи елемент десно од a_j строго мањи од a_j – скуп

За сваки интервал (a_i, a_j), где је a_i минимум префикса $a_0, \ldots, a_j$ треба проверити да ли у десном делу низа, иза позиције j, постоји елемент a_k који припада том интервалу. Постоје два приступа да се то ефикасно урадимо:

- Потребно и довољно да десно од a_j постоји елемент који је строго мањи од a_j и да за највећи елемент десно од a_j који је строго мањи од a_j важи да је строго већи од a_i.
- Потребно је и довољно да десно од a_j постоји елемент који је строго већи од a_i и да за најмањи елемент десно од a_j који је строго већи од a_i важи да је строго мањи од a_j.

Илуструјмо решење засновано на првом приступу. Тврђење важи, јер ако је највећи елемент десно од a_j који је строго мањи од a_j мањи или једнак од a_i, онда су сви елементи десно од a_j строго мањи од a_j мањи или једнаки a_i и ниједан не припада интервалу (a_i, a_j).

Једно ефикасно решење можемо добити коришћењем структуре података у којој ћемо чувати све елементе десно од текућег елемента a_j, а која нам омогућава да међу њима ефикасно пронађемо највећи елемент који

је строго мањи од дате вредности a_j. У језику C++ можемо користити библиотечку колекцију скуп (`set`) уређен опадајући и њену методу `upper_bound` (ефикасна метода `upper_bound`, која ради у логаритамској сложености, постоји код уређених скупова `set`, али не и код неуређених скупова `unordered_set`).

Елементе a_j можемо обилазити здесна налево, за сваког проверавати да ли је средишњи елемент неке 132-тројке и ако није, додавати га у скуп (јер ће он бити десно од свих елемената које ћемо у наредним итерацијама обрађивати). Обиласком слева надесно требало би елементе избацивати из скупа, што је мало неелегантније.

Обилазак низа здесна налево мало компликује проналажење минимума префикса (који се израчунавају инкрементално, слева надесно, међутим, њих можемо одредити у посебном пролазу на самом почетку и сместити у помоћни низ (слично као у задатку Пар производа у ранцу).

Додавање елемената у скуп који садржи m елемената, као и одређивање највећег елемента већег од дате вредности су сложености $O(m)$. Пошто се и додавање и претрага врше n пута, сложеност овог приступа је $O(n \log n)$ (израчунавање максимума префикса које се врши у фази претпроцесирања је сложености $O(n)$).

```
#include <iostream>
#include <vector>
#include <set>
#include <stack>
#include <functional>

using namespace std;

bool svojstvo132(const vector<int>& a) {
  int n = a.size();
  // niz minimuma prefiksa - na poziciji i nalazi se minimum prefiksa
  // niza na pozicijama [0, i]
  vector<int> minP(n);
  minP[0] = a[0];
  for (int i = 1; i < n; i++)
    minP[i] = min(minP[i-1], a[i]);

  // elementi desno od aj
  set<int, greater<int>> elementi_desno;
  elementi_desno.insert(a[n-1]);

  // za svaki element aj proveravamo da li moze biti sredisnji u nekoj
  // 132-trojci
  for (int j = n-2; j > 0; j--) {
    // analiziramo interval (ai, aj) i proveravamo da li postoji ak
    // desno od j koji mu pripada
    int ai = minP[j], aj = a[j];
    // interval je prazan
    if (ai == aj)
      continue;
    // trazimo najveci element desno od aj koji je strogo manji od aj
    auto najveci_desno_manji_od_aj = elementi_desno.upper_bound(aj);
    // ako on postoji i strogo je veci od ai, pronadjena je 132-trojka
    if (najveci_desno_manji_od_aj != elementi_desno.end() &&
        *najveci_desno_manji_od_aj > ai)
      return true;

    // najveci element desno od aj koji je manji od aj je manji ili
    // jednak ai, pa aj ne moze biti sredisnji element trojke

    // aj ce biti desno od elemenata u narednim iteracijama
    elementi_desno.insert(aj);
  }
  // nije pronadjena 132-trojka
```

```
  return false;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << (svojstvo132(a) ? "da" : "ne") << endl;
  return 0;
}
```

Најмањи елемент десно од a_j строго већи од a_i – ред са приоритетом

За сваки интервал (a_i, a_j), где је a_i минимум префикса $a_0, \ldots, a_j$ треба проверити да ли у десном делу низа, иза позиције j постоји елемент a_k који припада том интервалу.

Постоје два приступа да се то ефикасно урадимо:

- Потребно и довољно да десно од a_j постоји елемент који је строго мањи од a_j и да за највећи елемент десно од a_j који је строго мањи од a_j важи да је строго већи од a_i.
- Потребно је и довољно да десно од a_j постоји елемент који је строго већи од a_i и да за најмањи елемент десно од a_j који је строго већи од a_i важи да је строго мањи од a_j.

Илуструјмо решење засновано на другом приступу. Тврђење важи, јер ако је најмањи елемент десно од a_j који је строго већи од a_i већи или једнак од a_j, онда су сви елементи десно од a_j строго већи од a_i већи или једнаки a_j и ниједан не припада интервалу (a_i, a_j).

Све кандидате за десни елемент a_k ћемо чувати у посебној колекцији (структури података) у којој ће се налазити само елементи десно од a_j који су строго већи од текуће вредности a_i. Испоставља се да је погодније обилазити елементе a_j здесна налево. Наиме, приликом обиласка низа здесна налево минимуми префикса се могу само повећавати, што значи да једном елиминисане елементе a_k (јер су мањи или једнаки од a_i) не треба више никада поново разматрати (ако су мањи или једнаки од a_i они ће бити мањи или једнаки и од неког новог минимума $a'_i > a_i$). Дакле, у колекцији чувамо само вредности десно од a_j које су строго веће од текућег минимума a_i, а повећавањем текућег минимума a_i све елементе мање или једнаке од њега избацујемо из колекције. Ако је након тога колекција непразна, упоређујемо њен најмањи елемент a_k са a_j и ако је он строго већи од a_j, пронађено је је 132-тројка. У супротном убацујемо елемент a_j у колекцију и прелазимо на претходни елемент a_j.

Очигледно је да структура података треба да буде таква да ефикасно можемо очитати и избацити најмањи елемент. За то можемо употребити скуп или, још боље, ред са приоритетом.

Обилазак низа здесна налево мало компликује проналажење минимума префикса (који се израчунавају инкрементално, слева надесно, међутим, њих можемо одредити у посебном пролазу на самом почетку и сместити у помоћни низ (слично као у задатку Пар производа у ранцу).

Приметимо да чињеница да је низ минимума префикса сортиран омогућава да неке елементе десно од a_j заувек елиминишемо из колекције у којој чувамо елементе десно од a_j, јер знамо да никада више неће моћи да буду део интервала (a_i, a_j). То нам омогућава да најмањи елемент десно од a_j строго већи од a_i тражимо само на почетку колекције (скупа или реда са приоритетом). Са друге стране низ средишњих елемената a_j није сортиран, па се у дуалном приступу у ком се тражи највећи елемент десно од a_j који је мањи од a_j ниједан елемент не може елиминисати из колекције и потребна нам је колекција (скуп) у којој се тражени елемент не мора наћи на почетку колекције, већ може бити на произвољном месту у њеној унутрашњости (таква претрага је принципијелно компликованија).

Пошто се уметање и брисање из реда са приоритетом који садржи m елемената извршава у сложености $O(\log m)$ и пошто сваки елемент може највише једном бити убачен и избачен из реда, сложеност овог решења је $O(n \log n)$ (израчунавање низа минимума префикса врши се током претпроцесирања у сложености $O(n)$).

```
#include <iostream>
#include <vector>
```

```cpp
#include <queue>
#include <functional>

using namespace std;

bool svojstvo132(const vector<int>& a) {
  int n = a.size();
  // niz minimuma prefiksa - na poziciji i nalazi se minimum prefiksa
  // niza na pozicijama [0, i]
  vector<int> minP(n);
  minP[0] = a[0];
  for (int i = 1; i < n; i++)
    minP[i] = min(minP[i-1], a[i]);

  // elementi desno od aj koji su kandidati za ak, sortirani neopadajuce
  priority_queue<int, vector<int>, greater<int>> elementi_desno;
  elementi_desno.push(a[n-1]);

  for (int j = n-2; j > 0; j--) {
    // analiziramo interval (ai, aj) i proveravamo da li postoji ak
    // desno od j koji mu pripada
    int ai = minP[j], aj = a[j];
    // interval je prazan
    if (ai == aj)
      continue;
    // uklanjamo elemente manje ili jednake od ai, jer oni nisu
    // kndidati (ne mogu da pripadnu ni trenutnom ni buducim
    // intervalima (ai, aj), jer se ai samo mogu povecati iduci nalevo)
    while (!elementi_desno.empty() && elementi_desno.top() <= ai)
      elementi_desno.pop();
    // pronadjena je trojka (ai, aj, ak)
    if (!elementi_desno.empty() && elementi_desno.top() < aj)
      return true;

    // najmanji element desno od aj koji je veci od ai je veci ili
    // jednak aj, pa aj ne moze biti sredisnji element trojke

    // aj ce biti desno od elemenata u narednim iteracijama pa je
    // kandidat za ak
    elementi_desno.push(aj);
  }

  // nije pronadjena 132-trojka
  return false;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << (svojstvo132(a) ? "da" : "ne") << endl;
  return 0;
}
```

Види другачија решења овог задатка.

9.4 Монотони стек и ред

Задатак: Најближи већи претходник

Познат је низ висина зграда поређаних дуж једне улице. Спајдермен жели да постави хоризонтално уже на сваку зграду које ће га довести до неке претходне зграде. Уже се може закачити било где на зграду (осим на кров). За сваку зграду је потребно исписати висину њој претходне зграде до које ће водити хоризонтално уже са текуће зграде (то је прва зграда лево од текуће чија је висина строго већа од текуће).

Улаз: Са стандардног улаза се уноси број зграда n ($1 \leq n \leq 50000$), а затим n висина зграда раздвојених размацима.

Излаз: На стандардни излаз исписати тражени низ висина зграда раздвојен размацима. Ако испред неке зграде не постоји строго виша зграда, исписати `-`.

Пример

Улаз	*Излаз*
7 2 7 2 4 1 3 6	- - 7 7 4 4 7

Решење

Нагласимо да нема никакве значајне разлике да ли се тражи најближи већи претходник, најближи мањи претходник, најближи претходник који је мањи или једнак или најближи претходник који је већи или једнак текућем елементу. Исти алгоритам се може применити на било коју релацију поретка.

Груба сила

Наивно решење засновано на линеарној претрази у ком би се за сваки елемент редом уназад тражио њему најближи већи претходник би било веома неефикасно (сложеност би му била $O(n^2)$ и тај најгори случај би се јављао код неопадајућих низова).

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  for (int i = 0; i < n; i++) {
    bool nadjen = false;
    for (int j = i-1; j >= 0; j--)
      if (a[j] > a[i]) {
        cout << a[j] << " ";
        nadjen = true;
        break;
      }
    if (!nadjen)
      cout << "- ";
  }
  cout << endl;

  return 0;
}
```

Рекурзивно решење

Покушајмо да ефикаснији алгоритам конструишемо индуктивно-рекурзивном конструкцијом.

- Базу чини једночлан низ и сигурни смо да почетни елемент нема претходника већег од себе (јер уопште нема претходника). Заиста, са прве зграде није могуће развући канап.
- Претпоставимо да за сваки елемент низа дужине k умемо да одредимо најближег већег претходника и размотримо како бисмо одредили најближег већег претходника последњег елемента у низу дужине $k+1$ (елемента на позицији k). Анализу крећемо од директног претходника текућег елемента (за позицију k, анализу крећемо од позиције $k-1$). Ако је тај елемент строго већи од текућег, он му је најближи већи претходник (и до њега развлачимо канап). У супротном рекурзивно одређујемо његовог најближег већег претходника и тако добијени елемент упоређујемо са текућим елементом. Тај поступак понављамо све док не дођемо до елемента који је већи од текућег елемента или до ситуације у којој неки елемент мањи или једнак од текућег нема већих претходника.

Овај алгоритам можемо рекурзивно изразити на следећи начин.

```
#include <iostream>
#include <vector>
using namespace std;

int najblizi_veci_prethodnik(const vector<int>& a, int k) {
  if (k == 0)
    return -1;
  int p = k-1;
  while (p != -1 && a[p] <= a[k])
    p = najblizi_veci_prethodnik(a, p);
  return p;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  for (int k = 0; k < n; k++) {
    int p = najblizi_veci_prethodnik(a, k);
    if (p == -1)
      cout << "-" << " ";
    else
      cout << a[p] << " ";
  }
  cout << endl;
  return 0;
}
```

Динамичко програмирање

Ова имплементација је неефикасна јер долази до вишеструког позивања функције за исте аргументе. Ствар се може поправити тако што применимо технику динамичког програмирања и памтимо вредности рекурзивних позива у низу.

Можда изненађујуће јер имплементација садржи угнежђене петље, сложеност најгорег случаја имплементације засноване на динамичком програмирању је линеарна тј. $O(n)$, но то није уопште једноставно доказати. Кључни аргумент који ће нам помоћи и да упростимо имплементацију и да лакше анализирамо сложеност је то да током извршавања многи елементи низа престају да буду кандидати за најближе веће претходнике, па се кроз њих никада не итерира током унутрашње петље `while`. Наиме, када се након неког елемента x појави неки елемент y који је већи од њега или му је једнак, тада елемент x престаје да буде кандидат за најближег већег претходника свих елемената који се јављају у низу иза y (јер је свим елементима z иза y елемент y увек ближи од елемента x, а ако је x строго већи z и y ће бити строго већи од z, јер је $x \leq y$). Заиста, ако се иза неке зграде појави зграда исте висине или виша од ње она ће заклонити претходну зграду и ниједан канап надаље неће моћи бити развучен до те ниже зграде. Стога се током петље `while` итерира кроз релативно

мали број кандидата. У сваком тренутку елементи који су кандидати за најближе мање предходнике чине опадајући низ. Сваки наредни елемент елиминише оне текуће кандидате који су мањи од њега или су му једнаки (свака нова зграда сакрива зграде које су ниже од ње или су јој исте висине). Први елемент у низу потенцијалних кандидата који је строго већи од текућег елемента је уједно његов најближи већи претходник. Ако такав елемент не постоји, онда елемент нема већег претходника.

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  vector<int> dp(n);
  dp[0] = -1;
  for (int k = 0; k < n; k++) {
    int p = k-1;
    while (p != -1 && a[p] <= a[k])
      p = dp[p];
    dp[k] = p;
    if (p == -1)
      cout << "-" << " ";
    else
      cout << a[p] << " ";
  }
  cout << endl;
  return 0;
}
```

Оптимизација помоћу стека

Пошто је низ кандидата опадајући, кандидати који бивају елиминисани (тј. заклоњени) се могу налазити само на крају низа потенцијалних кандидата. Ово указује на то да се текући кандидати за најближег већег претходника могу чувати на стеку и да није потребно истовремено чувати читав низ `najblizi_veci_prethodnik`, већ само оне кандидате који нису елиминисани јер су заклоњени неким већим или једнаким елементом (на стеку можемо чувати било позиције тих кандидата, било њихове вредности). У тренутку када се обрађује елемент на позицији i на стеку се налазе вредности или позиције i такве да је a_i максимум елемената низа a на позицијама из интервала $[i, j)$. Овим добијамо имплементацију веома сличну претходној, и са истом асимптотском меморијском сложеношћу најгорег случаја. Ипак, за многе улазе, заузеће меморије ће бити мање (ако на стеку чувамо вредности, не морамо чак ни да памтимо цео оригинални низ).

Прикажимо рад овог алгоритма на примеру низа `3 1 2 5 3 1 4`.

На почетку је стек празан.

Пошто је стек празан, `3` нема строго већих претходника. Након тога се он додаје на стек.

```
3
```

Пошто је елемент `1` мањи од елемента `3` који је на врху стека, елемент `3` је његов најближи већи претходник. Након тога се на стек додаје `1`.

```
3 1
```

Када на ред дође елемент `2` са стека се скида елемент `1` који је мањи од њега и проналази се да је најближи већи претходник броја `2` број `3`. Елемент `1` заиста престаје да буде кандидат свим каснијим елементима (зграда висине `2` заклања зграду висине `1`) и елемент `2` који се поставља на стек преузима његову улогу.

```
3 2
```

Када на ред дође елемент 5 са стека се скидају елементи 3 и 2. Елементи 3 и 2 заиста престају да буду кандидати свим каснијим елементима (зграда висине 5 заклања зграде висине 3 и 2). Пошто је стек празан, елемент 5 нема већих претходника. Елемент 5 се поставља на стек.

5

Пошто је елемент 3 мањи од елемента 5 који је на врху стека, елемент 5 је његов најближи већи претходник. Након тога се на стек додаје 3.

5 3

Пошто је елемент 1 мањи од елемента 3 који је на врху стека, елемент 3 је његов најближи већи претходник. Након тога се на стек додаје 1.

5 3 1

Када на ред дође елемент 4 са стека се скидају елементи 1 и 3 који су мањи од њега и проналази се да је најближи већи претходник броја 4 број 5. Елементи 3 и 1 заиста престају да буду кандидати свим каснијим елементима (зграда висине 4 заклања зграде висине 1 и 3) и елемент 4 који се поставља на стек преузима њихову улогу.

У језику C++ стек се нуди кроз колекцију `stack`. Елементе додајемо методом `push`, елемент на врху стека можемо очитати методом `top`, а елемент са врха можемо уклонити методом `pop` (она не враћа вредност). Проверу да ли је стек празан можемо извршити методом `empty`.

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  stack<int> s;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    while (!s.empty() && s.top() <= x)
      s.pop();
    if (s.empty())
      cout << "-" << " ";
    else
      cout << s.top() << " ";
    s.push(x);
  }
  cout << endl;
  return 0;
}
```

Стек можемо имплементирати и ручно, уз помоћ низа.

```
#include <iostream>

using namespace std;

const int MAX = 50000;

int main() {
  int n;
  cin >> n;

  int s[MAX];
```

```
  int sp = 0;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    while (sp > 0 && s[sp - 1] <= x)
        sp--;
    if (sp == 0)
      cout << "-" << " ";
    else
      cout << s[sp - 1] << " ";
    s[sp++] = x;
  }
  cout << endl;

  return 0;
}
```

Задатак: Најближи већи следбеник

Познат је низ висина зграда поређаних дуж једне улице. Спајдермен жели да постави хоризонтално уже на сваку зграду које ће га довести до неке наредне зграде. Уже се може закачити било где на зграду (осим на кров). За сваку зграду је потребно исписати висину њој најближе наредне зграде до које ће водити хоризонтално уже са текуће зграде (то је прва зграда десно од текуће чија је висина строго већа од текуће).

Улаз: Са стандардног улаза се уноси број зграда n ($1 \leq n \leq 50000$), а затим, у наредној линији, n висина зграда раздвојених размацима.

Излаз: На стандардни излаз исписати тражени низ висина зграда раздвојен размацима. Ако иза неке зграде не постоји строго виша зграда, исписати `-`.

Пример

Улаз	*Излаз*
`9` `1 3 3 5 4 2 6 1 2`	`3 5 5 6 6 6 - 2 -`

Задатак: Највећи правоугаоник у хистограму

Напиши програм који одређује површину највећег правоугаоника који се може уписати у задати хистограм (хистограм се састоји од стубаца ширине 1).

Улаз: Са стандардног улаза се уноси број стубаца n ($1 \leq n \leq 50000$), а затим и висине стубаца (позитивни цели бројеви, раздвојени размацима).

Излаз: На стандардни излаз исписати тражену површину.

Пример

Улаз	*Излаз*
`6` `3 7 4 6 5 8`	`20`

Задатак: Највећи квадрат у хистограму

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Задатак: Збир минимума

Напиши програм који одређује збир минимума свих непразних сегмената датог низа целих бројева.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$), а затим и n елемената низа (бројева између 0 и 10^5) раздвојених са по једним размаком).

Излаз: Пошто збир може бити велики, на стандардни излаз исписати његов остатак са бројем милијарду и седам ($10^9 + 7$).

Пример 1

Улаз	*Излаз*
3 1 3 2	10

Објашњење

Непразни сегменти низа су $[1]$, $[1,3]$, $[1,3,2]$, $[3]$, $[3,2]$ и $[2]$. Њихови минимуми су редом 1, 1, 1, 3, 2 и 2, а збир тих минимума је 10.

Пример 1

Улаз

```
7
4 3 2 1 2 3 4
```

Излаз

```
48
```

Пример 2

Улаз

```
7
1 2 3 3 3 2 1
```

Излаз

```
49
```

Задатак: Сегменти оивичени максимумима

Одредити колико постоји сегмената (бар двочланих поднизова узастопних елемената низа) у низу различитих бројева у којима су сви елементи унутар сегмента строго мањи од елемената на њиховим крајевима.

Улаз: Са стандардног улаза се уноси број n, а затим n различитих природних бројева (сваки у посебној линији).

Излаз: На стандардни излаз испиши тражени број сегмената.

Пример

Улаз	*Излаз*
8 5 3 8 4 2 7 10 6	11

Задатак: Својство 132

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Монотони стек

Пажљивом анализом можемо још мало поправити сложеност и упростити решење тако што ћемо уместо реда са приоритетом или скупа употребити стек у коме ће сви кандидати за вредност a_k бити уређени у неопадајућем редоследу (на врху стека налазиће се најмањи). Могуће је једноставно елиминисати дупликате и постићи растући редослед, али то ништа значајно не мења.

Приликом анализе сваког пара (a_i, a_j), где је a_i минимум префикса који се завршава елементом a_j, из колекције треба заувек избацити све елементе a_k који су мањи или једнаки од a_i (јер они не припадају нити текућем, нити наредним интервалима (a_i, a_j)). Пошто су елементи на стеку сортирани неопадајуће (или чак растуће), од врха ка дну, ти елементи се могу налазити само на врху стека и лако ћемо их детектовати и уклонити. Ако након тога стек остане непразан, на његовом врху ће бити елемент a_k, који је најмањи од свих елемената десно од a_j, који су строго већи од a_i. Ако је a_k строго мањи од a_j (тј. ако је $a_k < a_j$), онда a_k припада интервалу (a_i, a_j) и пронађена је 132-тројка. У супротном знамо да је a_j мањи или једнак од елемента a_k на врху стека (тј. да је $a_k \geq a_j$) и приликом додавања елемента a_j на стек, њега треба додати на врх, чиме стек остаје сортиран неопадајуће (ако је елемент $a_j = a_k$, можемо прескочити додавање елемента a_j и тиме избећи непотребно чување копија истог елемента и постићи да је стек увек сортиран растуће).

Пошто су операције читања, уклањања и додавања елемента на врх стека константне сложености, а сваки елемент највише једном може бити додат и сколњен са стека, укупна сложеност рада са стеком је $O(n)$. Пошто се и израчунавање минимума префикса врши у сложености $O(n)$, укупна сложеност овог решења је $O(n)$.

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

bool svojstvo132(const vector<int>& a) {
  int n = a.size();
  // niz minimuma prefiksa - na poziciji i nalazi se minimum prefiksa
  // niza na pozicijama [0, i]
  vector<int> minP(n);
  minP[0] = a[0];
  for (int i = 1; i < n; i++)
    minP[i] = min(minP[i-1], a[i]);

  // elementi desno od aj koji su kandidati za ak
  stack<int> elementi_desno;
  elementi_desno.push(a[n-1]);
  // za svaki element aj proveravamo da li moze biti sredisnji u nekoj
  // 132-trojci
  for (int j = n-2; j > 0; j--) {
    // analiziramo interval (ai, aj) i proveravamo da li postoji ak
    // desno od j koji mu pripada
    int ai = minP[j], aj = a[j];
    // interval je prazan
    if (aj == ai)
      continue;
    // uklanjamo elemente manje ili jednake od ai, jer oni nisu
    // kndidati (ne mogu da pripadnu ni trenutnom ni buducim
    // intervalima (ai, aj), jer se ai samo mogu povecati iduci nalevo)
    while (!elementi_desno.empty() && elementi_desno.top() <= ai)
      elementi_desno.pop();
    // pronadjena je trojka (ai, aj, ak)
    if (!elementi_desno.empty() && elementi_desno.top() < aj)
      return true;

    // najmanji element desno od aj koji je veci od ai je veci ili
    // jednak aj, pa aj ne moze biti sredisnji element trojke

    // aj ce biti desno od elemenata u narednim iteracijama pa je
    // kandidat za ak
    elementi_desno.push(aj);
  }
```

```
  // nije pronadjena 132-trojka
  return false;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << (svojstvo132(a) ? "da" : "ne") << endl;
  return 0;
}
```

Задатак: Max сегмената дужине k

Дат је низ целих бројева и природан број k. Написати програм којим се приказују редом максимални елементи свих сегмента дужине k (сегмент чине узастопни елементи низа).

Улаз: У првој линији налази се природан број k ($0 < k \leq n$) дужине сегмената. Наредна линија стандардног улаза садржи природан број n ($2 \leq n \leq 50000$), број елемената низа. У свакој од n наредних линија стандардног улаза, налази по један члан низа.

Излаз: За сваки сегмент дужине k на стандардном излазу приказати максимални елемент, сваки у посебној линији.

Пример

Улаз	*Излаз*
3	3
7	3
1	4
2	5
3	5
1	
4	
5	
2	

Решење

Груба сила

Задатак можемо решити анализирајући све сегменте дужине k и за сваки одредити максимум. Спољашњим циклусом дефишемо почетак сваког сегмента (i узима вредности од 0 до $n-k$) а у унутрашњем цилусу (j узима вредности од 0 до $k-1$) пролазимо по свих k елемената сегмента који почиње на позицији i и налазимо максимални елемент. На тај начин вршимо око $(n-k+1) \cdot k$ провера. У најгорем случају, када је k око $\frac{n}{2}$, то је $O(n^2)$ операција.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int k;
  cin >> k;
  int n;
  cin >> n;
  vector<int> a(n);
```

```
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // za svaki pocetak segmenta duzine k
  for (int i = 0; i + k - 1 < n; i++) {
    // izracunavamo i ispisujemo maksimum segmenta a[i, i+k-1]
    int max = a[i];
    for (int j = i + 1; j < i + k; j++) {
      if (a[j] > max)
        max = a[j];
    }
    cout << max << endl;
  }
  return 0;
}
```

Инкрементално решење

Два узастопна сегмента дужине k имају $k-1$ заједничких елемената и разликују се само за први елемент првог и последњи елемент наредног сегмента. Покушајмо да конструишемо инкрементални алгоритам који ће искористити ову чињеницу. Прва идеја је да максимум новог сегмента одредимо на основу вредности максимума првог сегмента, елемента који се из њега уклања и елемента који се у њега додаје приликом преласка на нови сегмент. Уколико је је први елемент мањи од максимума, тада се максимум новог сегмента може добити као максимум старог максимума и новог елемента. Међутим, ако је максимум сегмента баш први елемент који се избацује и уколико је елемент који се додаје мањи од њега, онда немамо баш никакву информацију о томе шта је максимум новог сегмента (то може бити елемент који се додаје, или било који од $k-1$ заједничких елемената). Стога је потребно проширити наше знање о сваком сегменту.

Памћење низа максимума свих суфикса

Ако знамо максимум свих елемената иза првог елемента сегмента, онда се максимум проширеног сегмента може лако израчунати поређењем те вредности и вредности новог елемента. Међутим, да бисмо за тај сегмент могли знати максимум његовог суфикса без првог елемента (а то нам је неопходно због наредног проширења), потребно је да за први сегмент знамо и максимум суфикса без прва два његова елемента. Ово се наставља и заправо је потребно да за сваки сегмент знамо максимуме свих суфикса. На пример, ако је сегмент $5, 8, 4, 2, 3, 6$, и ако је $k = 4$, тада је потребно да у почетку знамо редом максимуме сегмената $5, 8, 4, 2$, затим $8, 4, 2$, затим $4, 2$ и на крају 2. То ће бити редом вредности $8, 8, 4, 2$. Преласком на наредни сегмент $8, 4, 2, 3$, можемо да искористимо то што знамо претходне максимуме и да на основу њих добијемо вредности максимума сегмената $8, 4, 2, 3$, затим $4, 2, 3$, затим $2, 3$ и на крају 3. То ће бити редом вредности $8, 4, 3, 3$. Након тога прелазимо на сегмент $4, 2, 3, 6$ и максимуме вредности сегмената $4, 2, 3, 6$, затим $2, 3, 6$, затим $3, 6$ и на крају 6 одређујемо као $6, 6, 6, 6$. Анализирајмо како смо знајући серију максимума свих суфикса сегмента могли да одредимо серију максимума свих суфикса новог сегмента. Сви суфикси новог сегмента се добијају дописивањем последњег његовог елемента на све суфиксе полазног сегмената изузев првог (оног једнаког целом полазном сегменту). Имајући ово у виду сваки од максимума суфикса новог сегмента можемо изградити тако што узмемо већу од вредности максимума одговарајућег суфикса полазног сегмента и броја који се додаје (последњи суфикс као и његов максимум једнак је броју који се додаје). На пример, ако су максимуми суфикса полазног сегмента били $8, 8, 4, 2$ и ако је нови елемент 3, онда су максимума суфикса новог сегмента одређени тако што занемаримо први елемент (8), затим редом узимамо веће од вредности $8, 4, 2$ и броја 3 (чиме добијамо $8, 4, 3$) и на крају додамо број 3 (што даје решење $8, 4, 3, 3$).

Ако у сваком кораку ажурирамо максимуме свих суфикса, добијамо опет алгоритам који извршава $(n-k+1) \cdot k$ операција.

Памћење максимума карактеристичних суфикса

Претходни алгоритам је могуће значајно оптимизовати, ако уместо максимума свих суфикса, памтимо само максимуме неких карактеристичних суфикса. Наиме, приметимо да се у низу максимума суфикса одређени елементи често понављају више пута. Заправо, елемент који је максимум неког суфикса ће сигурно бити максимални у свим суфиксима закључно са оним који почиње тим елементом. На пример, у сегменту $5, 7, 8, 3, 4$, вредност 8 је максимум суфикса $5, 7, 8, 3, 4$, затим $7, 8, 3, 4$ и на крају $8, 3, 4$. Након тога, вредност 4 максимум суфикса $3, 4$ и 4. Уместо да одржавамо низ максимума свих суфикса $8, 8, 8, 4, 4$, довољно је да одржавамо

низ вредности максимума оних суфикса чији максимуми нису одређени вредношћу неког претходног (ширег) суфикса. У овом примеру, довољно је запамтити низ $8, 4$. Ако, на пример посматрамо сегмент $3, 4, 3, 4, 2$ довољно је запамтити низ $4, 4, 2$. Наиме максимум суфикса $3, 4, 3, 4, 2$, као и $4, 3, 4, 2$, је 4, максимум суфикса $3, 4, 2$ исто као и суфикса $4, 2$ је опет 4 (захваљујући другом појављивању елемента 4, које они садрже), док је вредност максимума последњег суфикса 2. Приметимо да максимуми свих суфикса сегмента као и карактеристични максимуми чине нерастући низ.

Дакле, ако посматрамо сегмент $a_i, \ldots, a_{i+k-1}$, и низ максимума $a_{j_1}, \ldots, a_{j_m}$ тада је елемент a_{j_1} максимум целог сегмента и свих суфикса који почињу на позицијама испред првог појављивања елемента a_{j_1}, елемент a_{j_2} је максимум суфикса који почиње непосредно иза првог појављивања елемента a_{j_1} и свих суфикса који почињу на позицијама испред првог појављивања елемента a_{j_2}, елемент a_{j_3} је максимум суфикса који почиње иза првог појављивања елемента a_{j_2} и свих суфикса који почињу на позицијама испред првог појављивања елемента a_{j_3} итд.

Покажимо да је овакав низ максимума веома једноставно изградити и одржавати приликом инкременталне промене сегмента (што је довољно за решење задатка, јер се максимум текућег сегмента једноставно очитава као први елемент низа карактеристичних максимума). Сваки нови сегмент од претходног добијамо уклањањем почетног елемента и додавањем новог елемента на крај. Посматрајмо како то утиче на описани низ максимума.

Приликом уклањања првог елемента сегмента постоје две могућности.

- Размотримо сегмент $3, 4, 8, 5, 2$ и његов низ карактеристичних максимума $8, 5, 2$. Уклањањем елемента 3, сегмент постаје $4, 8, 5, 2$, максимум целог низа и суфикса $8, 5, 2$ је и даље 8, максимум суфикса $5, 2$ је и даље 5, а максимум суфикса 2 је и даље 2, тако да низ максимума остаје непромењен $8, 5, 2$. Дакле, ако је први елемент сегмента различит од првог елемента низа максимума, онда низ максимума треба да остане непромењен (први елемент низа је сигурно већи од избаченог елемента сегмента и он остаје максимум суфикса који се добије избацивањем првог елемента сегмента).
- Размотримо сада сегмент $8, 3, 4, 5, 2$ и његов низ карактеристичних максимума $8, 5, 2$. Уклањањем елемента 8, добијамо сегмент $3, 4, 5, 2$, чији је максимум 5, исто као и за суфиксе $4, 5, 2$ и $5, 2$. Елемент 2 је максимум суфикса 2. Тиме добијамо низ карактеристичних максимума $5, 2$. Дакле, ако је елемент који се избацује из сегмента једнак максимуму целог сегмента, онда је максимум скраћеног сегмента заправо други елемент низа (јер он представља максимум дела сегмента иза првог појављивања првог максимума, а то је управо део сегмента без почетног елемента). Даљи елементи низа остају непромењени. Дакле, када се из сегмента уклања почетни елемент који је једнак првом елементу низа, тада се нови низ максимума добија уклањањем првог елемента низа максимума.

Размотримо сада случај додавања новог елемента на крај сегмента.

- Посматрајмо сегмент $8, 3, 4, 5, 2$ и његов низ карактеристичних максимума $8, 5, 2$. Ако сегмент проширимо елементом 1, низ карактеристичних максимума је $8, 5, 2, 1$. Заиста, елемент 8 је и даље максимум суфикса $8, 3, 4, 5, 2, 1$, елемент 5 је максимум суфикса $3, 4, 5, 2, 1$, затим $4, 5, 2, 1$, и $5, 2, 1$, елемент 2 је максимум суфикса $2, 1$, а елемент 1 је максимум суфикса 1. Ако на крај сегмента $8, 3, 4, 5, 2$ додајемо елемент 4, тада је низ карактеристичних максимума $8, 5, 4$. Заиста, 8 је максимум суфикса $8, 3, 4, 5, 2, 4$, 5 је максимум суфикса $3, 4, 5, 2, 4$, затим $4, 5, 2, 4$, и $5, 2, 4$, а 4 је максимум суфикса $2, 4$ и 4. Дакле, са краја низа карактеристичних максимума уклањамо све оне елементе који су строго мањи од новог елемента (низ се том операцијом може и испразнити) и додајмо тај нови елемент на крај низа карактеристичних максимума.

  Докажимо са се овим поступком коректно ажурира низ карактеристичних максимума. Ако је ажурирања низ карактеристичних максимума једночлан, тада је почетни елемента старог низа максимума (који је био максимум целог старог сегмента) строго мањи од новог елемента, тако да је нови елемент максимум проширеног сегмента и он се јавља тек на последњој позицији, тако да је заиста он максимум свих суфикса новог низа. Претпоставимо да низ максимума након обраде садржи више елемената $a_{j_1}, \ldots, a_{j_m}, a_j$. Знамо да је a_j ново-додани елемент сегмента и важи да је $a_{j_m} \geq a_j$. Зато за свако l од 1 до m важи да је $a_{j_l} \geq a_{j_m} \geq a_j$, па је елемент a_{j_l} и даље максимум сегмента који почиње иза првог појављивања $a_{j_{l-1}}$ (специјално a_1 је и даље максимум целог сегмента). Размотримо суфикс који почиње на првој позицији иза последњег појављивања a_{j_m}. Ако је тај сегмент једночлан (ако је a_{j_m} последњи елемент старог сегмента), онда се у њему налази само a_j и он му је максимум. Ако a_{j_m} није последњи онда је у старом низу максимума иза њега постојао неки елемент $a_{j_{m+1}}$, међутим он је уклоњен па је $a_{j_{m+1}} < a_j$, што значи да је a_j стварно максимум суфикса новог сегмента који почиње на

првој позицији иза a_{j_m}.

Алгоритам започињемо од празног низа максимума и у првој фази, док не обрадимо први сегмент од k почетних елемената полазног низа, примењујемо само операцију проширивања сегмента. Након тога прелазимо у другу фазу у којој у сваком кораку примењујемо операцију избацивања елемента са почетка и додавања елемента на крај сегмента (и одговарајућег ажурирања низа максимума). У сваком кораку, први елемент низа максимума ће бити максимум текућег k-точланог сегмента и треба га исписати.

Прикажимо како алгоритам ради на примеру низа $8, 3, 4, 5, 2, 6, 7, 1, 5, 3$ и $k = 4$. У првој колони је приказан текући сегмент, у другој низ карактеристичних максимума, а у трећој максимум који се исписује.

```
8             8
8 3           8 3
8 3 4         8 4
8 3 4 5       8 5           8
3 4 5         5
3 4 5 2       5 2           5
4 5 2         5 2
4 5 2 6       6             6
5 2 6         6
5 2 6 7       7             7
2 6 7         7
2 6 7 1       7 1           7
6 7 1         7 1
6 7 1 5       7 5           7
7 1 5         7 5
7 1 5 3       7 5 3         7
```

Приметимо да се од свих обрађених елемената низа једини кандидати да постану максимуми неког сегмента они који се налазе у низу карактеристичних максимума. Када се у сегменту појави елемент који је већи од неких кандидата, они се уклањају и престају да буду кандидати.

Претходно описано решење је врло ефикасно, сваки елемент низа укључимо и искључимо из реда највише једанпут, па је број потребних операција ограничен вредношћу $2 \cdot n$ тј. сложености је $O(n)$. Приметимо да је за ефикасније решење био потребан додатни простор величине $O(k)$.

Приметимо да се низ карактеристичних максимума понаша као ред са два краја - елементи се уклањају и са почетка и са краја, а додају се увек на крај. Стога је за његову реализацију пожељно користити неку библиотечку имплементацију реда. У језику C++ најпогоднија је колекција `deque`.

```cpp
#include <iostream>
#include <deque>
#include <queue>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  int k;
  cin >> k;
  // ucitavamo ulazne podatke
  int n;
  cin >> n;

  // red u kome cuvamo trenutni segment
  queue<int> segment;

  // red karakteristicnih maksimuma tekuceg segmenta
  // - prvi element reda je maksimum tekuceg segmenta i svih njegovih
  //   sufiksa koji pocinju pre prvog pojavljivanja tog elementa u segmentu
  // - svaki naredni element reda je maksimum sufiksa koji pocinje
```

```
  // neposredno iza prvog pojavljivanja u segmentu njemu prethodnog
  // elementa u redu i svih njegovih sufiksa koji pocinju pre prvog
  // pojavljivanja tog elementa u segmentu
  deque<int> maksimumi;
  for (int i = 0; i < n; i++) {
    if (i >= k) {
      // Posle prvih k koraka pocinjemo da skracujemo tekuci segment
      int prvi = segment.front();
      segment.pop();
      // Ako je element koji se izbacuje jednak prvom elementu u redu
      // maksimuma, on se izbacuje iz reda.
      if (maksimumi.front() == prvi)
        maksimumi.pop_front();
    }

    // dodajemo novi element
    int ai;
    cin >> ai;
    segment.push(ai);

    // Azurira se red maksimuma
    while(!maksimumi.empty() && ai > maksimumi.back())
      maksimumi.pop_back();
    maksimumi.push_back(ai);

    // nakon sto je prvih k elemenata ubaceno u segment, pocinjemo da
    // u svakom koraku ispisujemo maksimume segmenata
    if (i >= k - 1)
      cout << maksimumi.front() << endl;
  }

  return 0;
}
```

Ред са два краја можемо и сами имплементирати (уз помоћ низа или вектора).

```
#include <iostream>
#include <vector>

using namespace std;

struct Red {
  vector<int> red;
  int pocetak;
  int kraj;
};

void napravi(Red& red, int k) {
  red.red.resize(k);
  red.pocetak = red.kraj = -1;
}

void dodajNaKraj(Red& red, int x){
  if (red.pocetak == -1)
    red.pocetak = red.kraj = 0;
  else if (red.kraj == red.red.size() - 1)
    red.kraj = 0;
  else
    red.kraj++;
```

```
  red.red[red.kraj] = x;
}

void brisiPoslednji(Red& red){
  if (red.kraj == red.pocetak)
    red.pocetak = red.kraj = -1;
  else if (red.kraj == 0)
    red.kraj = red.red.size() - 1;
  else
    red.kraj--;
}

void brisiPrvi(Red& red){
  if (red.kraj == red.pocetak)
    red.pocetak = red.kraj = -1;
  else if (red.pocetak == red.red.size() - 1)
    red.pocetak = 0;
  else
    red.pocetak++;
}

int prvi(const Red& red) {
  return red.red[red.pocetak];
}

int poslednji(const Red& red) {
  return red.red[red.kraj];
}

bool prazan(const Red& red) {
  return red.pocetak == -1;
}

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int k;
  cin >> k;
  int n;
  cin >> n;
  vector<int> a(n);
  for(int i = 0; i < n; i++)
    cin >> a[i];

  // red karakteristicnih maksimuma tekuceg segmenta
  // - prvi element reda je maksimum tekuceg segmenta i svih njegovih
  // sufiksa koji pocinju pre prvog pojavljivanja tog elementa u segmentu
  // - svaki naredni element reda je maksimum sufiksa koji pocinje
  // neposredno iza prvog pojavljivanja u segmentu njemu prethodnog
  // elementa u redu i svih njegovih sufiksa koji pocinju pre prvog
  // pojavljivanja tog elementa u segmentu
  Red red;
  napravi(red, k);
  for (int i = 0; i < n; i++) {
    // Posle prvih k koraka pocinjemo da skracujemo tekuci segment za
    // element a[i-k]. Ako je element koji se izbacuje jednak prvom
    // elementu u redu, on se izbacuje iz reda.
```

```
    if (i >= k && a[i - k] == prvi(red))
      brisiPrvi(red);
    // Segment se prosiruje elementu a[i] i u skladu sa tim se azurira
    // i red
    while(!prazan(red) && a[i] > poslednji(red))
      brisiPoslednji(red);
    dodajNaKraj(red, a[i]);

    // nakon sto je prvih k elemenata ubaceno u segment, pocinjemo da
    // u svakom koraku ispisujemo maksimume segmenata
    if (i >= k - 1)
      cout << prvi(red) << endl;
  }

  return 0;
}
```

Задатак: Најкраћи сегмент збира бар K

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Збирови префикса уз два одсецања - ред са два краја

Додатним одсецањем се програм може додатно поједноставити и решити веома ефикасно употребом реда са два краја или двоструко повезане листе. Наиме, ако се у низу префиксних збирова налазе бројеви z_j и $z_{j'}$, такви да је $j < j'$ и $z_j \geq z_{j'}$, тада је свако z_{i+1} (иза њих) довољно упоредити само са $z_{j'}$. Наиме, ако важи да је $z_{i+1} - z_j \geq K$, тада је и $z_{i+1} - z_{j'} \geq z_{i+1} - z_j \geq K$, међутим, дужина интервала $[j', i]$ је мања од дужине интервала $[j, i]$ (јер из $j < j'$ следи $i - j' + 1 < i - j + 1$). Зато се приликом додавања сваког префиксног збира у низ из низа могу елиминисати сви елементи који су већи или једнаки од њега. Приметимо да се на овај начин успоставља и одржава инваријанта да је низ збирова префикса увек сортиран по величини (сваки наредни елемент је строго већи од свих претходних). Стога се најмањи елементи увек налазе на почетку, а највећи елементи на крају, па се додавање увек врши само на крај, а уклањање са краја и почетка и уместо низа могуће је користити ред са два краја.

Приликом анализе сваког наредног збира z_{i+1} разматрају се редом елементи z_j са почетка реда. Сви за које важи $z_{i+1} - z_j \geq K$ се уклањају из реда (ажурирајући најмању дужину сегмента, ако је потребно). То се понавља све док се ред не испразни или док се не дође до елемента z_j таквог да је $z_{i+1} - z_j < K$.

Након тога се са краја реда уклањају сви елементи који су већи или једнаки од z_{i+1}. То се понавља све док се ред не испразни или док се на крају реда не појави неки елемент строго мањи од z_{i+1}. Тада се елемент z_{i+1} додаје на крај реда.

Пошто се сваки елемент највише једном може поставити у ред и уклонити из реда, а операције са редом су сложености $O(1)$, сложеност овог алгоритма је $O(n)$:

```
#include <iostream>
#include <deque>

using namespace std;

int main() {
  int k;
  cin >> k;
  int n;
  cin >> n;
  // duzina najkraceg segmenata (+beskonacno na pocetku)
  int minBrojElemenata = n + 1;
  // red u kom se cuvaju zbirovi prefiksa
  // za svaki zbir zj prvih j elemenata niza cuva se i broj j
```

```
  // red je uredjen rastuce
  deque<pair<int, int>> zbirovi;
  // zbir prefiksa [0, i]
  int zbir = 0;
  // zbir prvih 0 elemenata je 0
  zbirovi.emplace_back(zbir, 0);
  // analiziramo sve leve krajeve segmenta
  for (int i = 0; i < n; i++) {
    // ucitavamo element x = a[i] i uvecavamo zbir prefiksa
    int x; cin >> x;
    zbir += x;

    // analiziramo segmente [j, i] za dovoljno male vrednosti zbira zj
    while (!zbirovi.empty() && zbir - zbirovi.front().first >= k) {
      // azuriramo duzinu najkraceg segmenta
      int brojElemenata = i - zbirovi.front().second + 1;
      minBrojElemenata = min(minBrojElemenata, brojElemenata);
      // izbacujemo najmanji zbir iz reda
      zbirovi.pop_front();
    }

    // eliminisemo sve zbirove sa kraja reda koji su veci ili jednaki
    // od tekuceg zbira prvih i+1 elemenata niza
    while (!zbirovi.empty() && zbir <= zbirovi.back().first)
      zbirovi.pop_back();

    // dodajemo tekuci zbir prvih i+1 elemenata niza na kraj reda
    zbirovi.emplace_back(zbir, i+1);
  }

  // ispisujemo rezultat
  if (minBrojElemenata <= n)
    cout << minBrojElemenata << endl;
  else
    cout << "-" << endl;

  return 0;
}
```

У збирци се излажу напредније технике програмирања, које представљају увод у изучавање алгоритама и структура података. Централне теме ове збирке су коректност и сложеност алгоритама (укључујући анализу сложености и елементарне технике побољшања сложености алгоритама), рекурзивна и индуктивна конструкција алгоритама (укључујући технике попут "подели-па-владај", бектрекинга и динамчиког програмирања) и основне структуре података (скупови, мапе, стекови, редови, редови са приоритетом). Збирка подразумева основни нови знања из програмирања и предствља природан наставак "Методичке збирке задатака из основа програмирања". Познавање техника изложених у тој збирци представља предуслов за израду задатака у овој збирци.

Узраст ученика којима је збирка намењена није прецизно одређен. Збирка се може користити за додатну наставу и за припреме напредних нивоа такмичења у основној школи (државног и СИО), за додатну наставу и припрему основних нивоа такмичења у средњој школи (до државног), у редовној настави у другом разреду специјализованих ИТ одељења и електротехничких школа, као и у другом семестру изучавања програмирања тј. алгоритама и структура података на факултетима.