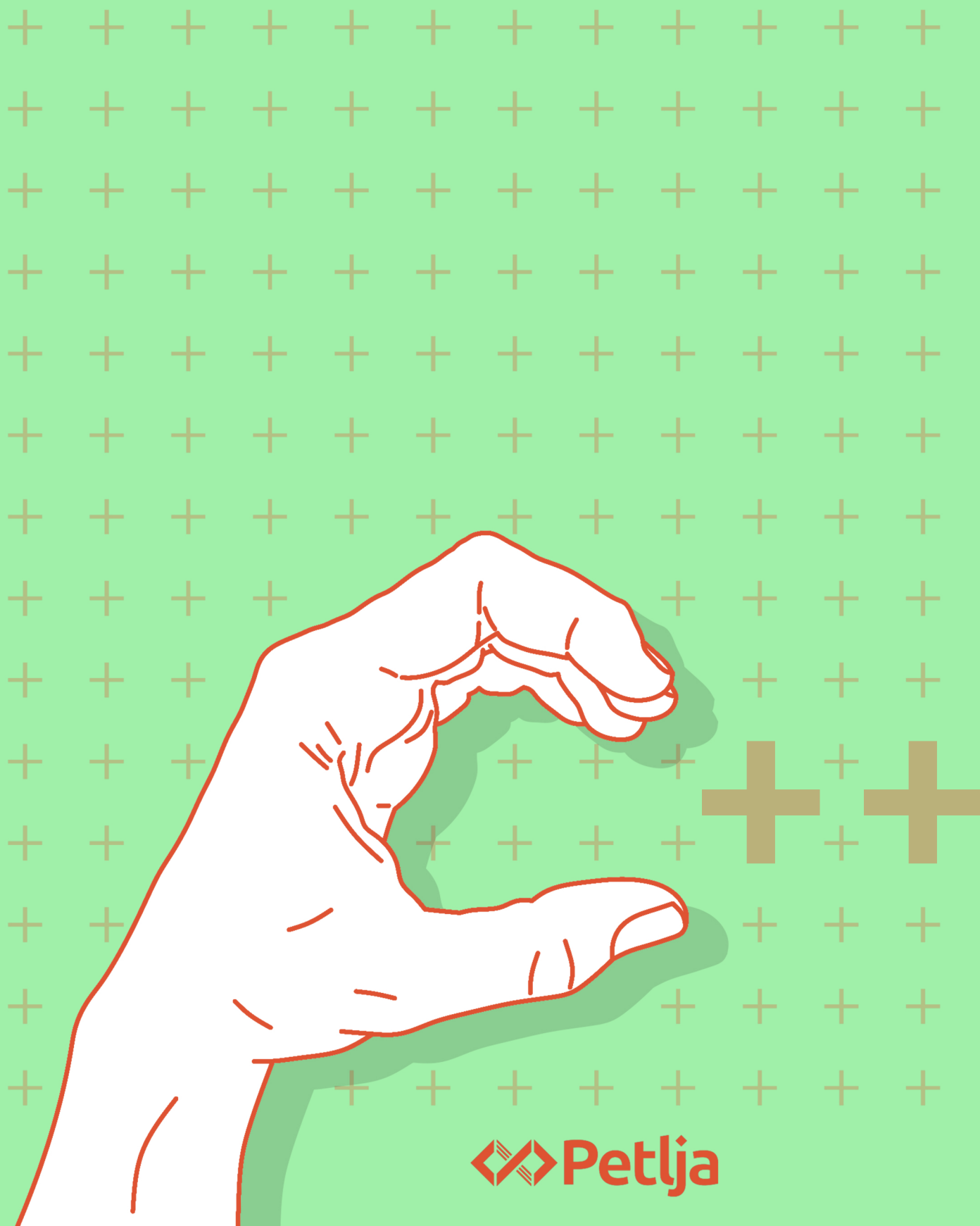


Методичка збирка задатака из алгоритмике - одабрани алгоритми и структуре података - C++



*Друштво математичара Србије
Фондација Пеиља*

Методичка збирка задатака из алгоритмике — C++

одабрани алгоритми и структуре података

Београд, 2019.

Аутори:

Филић Марић, професор на Математичком факултету у Београду
Весна Маринковић, доцент на Математичком факултету у Београду
Сјанка Мајиковић, професор у Математичкој гимназији у Београду
Душко Обрадовић, професор у гимназији „Вељко Петровић” у Сомбору
Милан Чабаркаја, професор у Математичкој гимназији у Београду

Методичка збирка задатака из алгоритмике — C++

одабрани алгоритми и структуре података

Издавачи: Друштво математичара Србије и Фондација „Петља”

За издавача: *др Војислав Андрић*

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

Издање: 1.1 (електронско)

Садржај

1	Графови	1
1.1	Репрезентација графа	1
1.2	Претрага у дубину и ширину	3
	Задатак: Достижни чворови	3
	Задатак: Пећине	6
	Задатак: Престононаследници	8
	Задатак: Пећине са тунелима	12
	Задатак: Компоненте повезаности	14
	Задатак: Чудна мрежа	15
	Задатак: Авионска преседања	17
1.3	Тополошко сортирање	19
	Задатак: Редослед послова	19
1.4	Најкраћи путеви од једног чвора	22
	Задатак: Најкраћи пут између два града	24
1.5	Најкраћи путеви између свих парова чворова	28
	Задатак: Сви најкраћи путеви у густом графу	29
	Задатак: Сви најкраћи путеви у ретком графу	35
1.6	Минимално повезујуће стабло	41
	Задатак: Уштеда каблова	41
	Задатак: Кластери	47
2	Алгебарски алгоритми	51
2.1	Модуларна аритметика	51
2.1.1	Сабирање и множење по модулу	52
2.1.2	Одузимање по модулу	52
2.1.3	Степеновање по модулу	52
	Задатак: Операције по модулу	52
	Задатак: Монопол	54
	Задатак: Разбрајалица	55
	Задатак: Збир бројева по модулу	56
	Задатак: Сат	57
	Задатак: Навијање сата	58
	Задатак: Степен по модулу	60
2.2	Теорија бројева	62
	Задатак: Савршени бројеви	62
	Задатак: Пријатељски бројеви	67
	Задатак: Прост број	69
	Задатак: Најближи прост број	72
	Задатак: Ератостеново сито	73
	Задатак: Збир простих прост	77
	Задатак: Велики прости бројеви	79
	Задатак: Број простих у интервалима	82
	Задатак: Расстављање на просте чиниоце	85
	Задатак: Допуна до пуног квадрата	88
	Задатак: Производи потпуни квадрати	90
	Задатак: Највећи заједнички делилац	92

	Задатак: Број дељив са 1 до n	97
	Задатак: Билијар из ћошка	98
	Задатак: Ученици на истим седиштима	100
	Задатак: Модуларни инверз	101
	Задатак: Кинеска теорема	107
	Задатак: Билијар	115
	Задатак: Ојлерова функција	118
2.3	Полиноми	121
	Задатак: Вредност полинома	121
	Задатак: Аритметика над полиномима	122
	Задатак: Множење полинома	124
3	Алгоритми текста	130
3.1	Тражење узорка у тексту	130
	Задатак: Префикс суфикс	130
	Задатак: Подниска	133
	Задатак: Подниске	136
	Задатак: Немењајуће ротације	137
	Задатак: Периодичност ниске	139
3.2	Најдужи палиндроми	141
	Задатак: Најдужа палиндромска подниска	141
	Задатак: Најкраћа допуна до палинрома	149
	Задатак: Најдужи подниз палиндром	151
3.3	Регуларни изрази	155
	Задатак: Датуми	155
3.4	Формалне граматике	156
	Задатак: Превођење потпуно заграђеног израза у постфиксни облик	156
	Задатак: Израз у коме нема заграда	159
	Задатак: Вредност израза	165
4	Геометријски алгоритми	170
4.1	Скаларни и векторски производ	170
	Задатак: Колинеарност и нормалност вектора	170
	Задатак: Колинеарне тачке	171
	Задатак: Са исте стране	173
	Задатак: Тачка у троуглу	174
	Задатак: Пресек дужи	176
4.2	Многоуглови	181
	Задатак: Конструкција простог многоугла	181
	Задатак: Припадност тачке конвексном многоуглу	185
	Задатак: Конвексни омотач	188
5	Напредне структуре података	194
5.1	Префиксно дрво	194
	Задатак: Днк секвенце	195
	Задатак: Коректни телефони	201
	Задатак: Днк префикси	204
	Задатак: Различите подниске	206
	Задатак: Пар који даје највећи XOR	208
	Задатак: Највећи XOR сегмента	211
5.2	Структура података за представљање дисјунктних подскупова (union-find)	214
	Задатак: Први пут кроз матрицу	218
	Задатак: Распоред са максималним збиром профита	221
5.3	Упити распона	224
5.3.1	Статички упити распона	224
	Задатак: Збирови сегмената	225
	Задатак: Увећавање сегмената	226
	Задатак: Пермутација са највећим збиром упита	229
5.3.2	Сегментна дрвета	229

	Задатак: Суме сегмената променљивог низа	231
5.3.3	Фенвикова стабла	236
	Задатак: Суме сегмената променљивог низа	238
	Задатак: Увећања сегмената и читање елемената	240
	Задатак: К-ти парни број	242
	Задатак: Трик са картама	245
	Задатак: Инверзије након избацивања сегмената	248
	Задатак: Број различитих елемената у сегментима	256
5.3.4	Лења сегментна дрвета	258
	Задатак: Увећања и зборови сегмената	259

Глава 1

Графови

Граф $G = (V, E)$ се састоји од скупа V чворова и скупа E грана. Најчешће грана одговара пару различитих чворова, мада су понекад дозвољене и петље, односно гране које воде од чвора ка њему самом. Граф може бити *неусмерен* или *усмерен*. Гране усмереног графа су уређени парови чворова и притом је важан редослед два чвора које повезује грана. Ако се граф представља цртежом, онда се гране усмереног графа цртају као стрелице усмерене од једног чвора (почетка) ка другом чвору (крају гране). Гране неусмереног графа су неуређени парови: оне се цртају као обичне дужи. *Степен* $d(v)$ чвора v је број грана суседних чвору v (односно број грана које чвор v повезују са неким другим чвором). У усмереном графу разликујемо *улазни степен* чвора v који је једнак броју грана за које је чвор v крај, односно *излазни степен* чвора v који је једнак броју грана за које је чвор v почетак.

Пут од чвора v_1 до чвора v_k је низ чворова $v_1, v_2, \dots, v_k$ повезаних гранама $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$. Пут је *једносмеран*, ако се сваки чвор у њему појављује само једном. За чвор u се каже да је *достигнут* из чвора v ако постоји пут (усмерен, односно неусмерен, зависно од графа) од чвора v до чвора u . По дефиницији је чвор v *достигнут* из v . *Циклус* је пут чији се први и последњи чвор поклапају. Циклус је *једносмеран* ако се, сем првог и последњег чвора, ни један други чвор у њему не појављује два пута. *Неусмерен облик* усмереног графа $G = (V, E)$ је исти граф, без смерова на гранама (тако да су парови чворова у E неуређени). За граф се каже да је *повезан* ако (у његовом неусмереном облику) постоји пут између произвољна два чвора. *Шума* је граф који (у свом неусмереном облику) не садржи циклусе. *Дрво* је повезана шума. *Коренско дрво* је усмерено дрво са једним посебно издвојеним чвором, који се зове *корен*.

Граф $H = (U, F)$ је *подграф* графа $G = (V, E)$ ако је $U \subseteq V$ и $F \subseteq E$. *Повезујуће дрво* неусмереног графа G је његов подграф који је дрво и садржи све чворове графа G . Повезујућа шума неусмереног графа G је његов подграф који је шума и садржи све чворове графа G . Ако неусмерени граф $G = (V, E)$ није повезан, онда се он може на јединствен начин разложити у скуп повезаних подграфа, који се зову *компоненте повезаности* графа G .

1.1 Репрезентација графа

Уобичајена су два начина представљања графова, Први је *матрица повезаности*, односно *матрица суседства* графа. Нека је $|V| = n$ и $V = \{v_1, v_2, \dots, v_n\}$. Матрица повезаности графа G је квадратна матрица $A = (a_{ij})$ реда n , са елементима a_{ij} који су једнаки 1 ако и само ако $(v_i, v_j) \in E$; остали елементи матрице A су нуле. Ако је граф неусмерен, матрица A је симетрична. Врста i ове матрице је дакле низ дужине n чија је j -та координата једнака 1 ако из чвора v_i води грана ка чвору v_j , односно 0 у противном. Недостатак матрице повезаности је то што она увек заузима простор величине n^2 , независно од тога колико грана има граф. Ако је број грана у графу мали, већина елемената матрице повезаности биће нуле. Ако се за представљање графа користи матрица повезаности, сложеност операције уклањања неке гране из графа је $O(1)$ и испитивање да ли су два чвора у графу повезана је такође $O(1)$.

У језику C++ можемо употребити следећу репрезентацију матрице.

```
bool A[MAX][MAX];
```

или, ако не знамо унапред број чворова тј. ако број чворова n сазнајемо тек у фази извршавања програма


```
vector<vector<bool>> A(n);
for (int i = 0; i < n; i++)
    A[i].resize(n);
```

Уместо да се и све непостојеће гране експлицитно представљају у матрици повезаности, могу се формирати повезане листе од јединица из i -те врсте, $i = 1, 2, \dots, n$. Овај други начин представљања графа зове се *листима повезаности*, односно *листима суседства*. Сваком чвору придружује се повезана листа, која садржи све гране суседне чвору (односно гране ка суседним чворовима). Листа може бити уређена према редним бројевима чворова на крајевима њених грана. Граф је представљен низом листа. Сваки елемент низа садржи име (индекс) чвора и показивач на његову листу суседа. Треба напоменути да иако име тако сугерише, имплементација овакве репрезентације графа не мора бити заснована на листама, већ се уместо повезаних листи може користити динамички низ или нека врста балансираних бинарних дрвета или пак хеш табела. У већини наредних алгоритама сматраћемо да је граф са којим радимо динамички и да је задат листом повезаности. У језку C++ можемо употребити следећу репрезентацију.

```
vector<vector<int>> susedi(n);
```

Нову грану можемо додати помоћу

```
susedi[cvor0d].push_back(cvorDo);
```

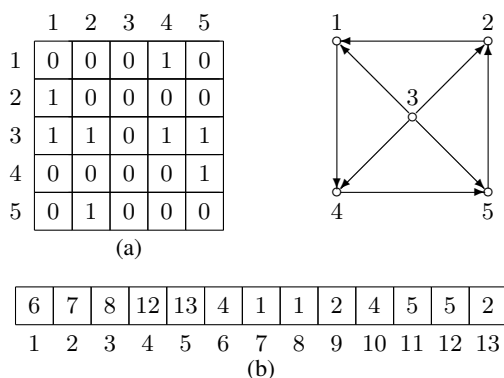
док итерацију кроз све суседне гране чвора можемо остварити помоћу

```
for (int cvorDo : susedi[cvor0d])
    ...
```

Оваква репрезентација омогућава да се у времену $O(1)$ нађу суседи сваког појединачног чвора у графу.

Матрице повезаности омогућавају да се у времену $O(1)$ испита да ли између два чвора постоји грана. С друге стране, листе повезаности омогућавају да се једноставније пронађу сви суседи неког датог чвора. Листе повезаности су меморијски ефикасније за графове са малим бројем грана (њихова меморијска сложеност је $O(V + E)$, за разлику од матрица повезаности чија је меморијска сложеност $O(V^2)$). У пракси се често ради са графовима који имају знатно мање грана од максималног могућег броја $(n(n-1)/2)$ неусмерених, односно $n(n-1)$ усмерених грана, и тада је ефикасније користити листе повезаности.

Ако је граф *сипајички*, односно нису дозвољена уметања и брисања, онда се цео граф може представити помоћу једног низа целих бројева, на следећи начин. У основи је и даље репрезентација у облику листа повезаности. Користи се низ дужине $|V| + |E|$. Првих $|V|$ чланова низа су придружени чворовима. Компонента низа придружена чвору v_i садржи индекс почетка списка чворова суседних чвору v_i , $i = 1, 2, \dots, n$. На слици 1.1 приказана на једном примеру је приказано представљање графа помоћу матрице повезаности и преко једног низа у коме се чувају све листе повезаности.



Слика 1.1: Представљање графа (a) матрицом повезаности, (b) једним низом у коме су представљене све листе повезаности.

Треба поменути да постоје и други начини за представљање графа, као што су матрице или листе инциденције где се за сваку грану чува информација са којим чворовима је инцидентна.

1.2 Претрага у дубину и ширину

Основни задаци над графом се често свode на то да се кренувши од неког почетног чвора и праћењем грана обиђу сви чворови графа који су достижни из тог полазног чвора. Ако је неусмерен граф повезан тиме ће се обићи сви његови чворови, а ако није, онда ће се обићи компонента повезаности којој припада почетни чвор. Код усмерених графова на овај начин ће се обићи компонента јаке повезаности којој припада почетни чвор.

Постоје два основна алгорита за обилазак графа: *претрага у дубину* и *претрага у ширину*.

Оба се могу представити следећом општом схемом.

```
dodaj pocetni cvor u kolekciju K
dok kolekcija K nije prazna:
    uzmi cvor c iz kolekcije K
    ako c nije oznacen:
        oznaci c
        za svaku granu cc'
            ubaci c' u kolekciju K
```

Означавањем чворова постижемо то да се алгоритам не упада у бесконачну петљу. Ако је колекција K стек, тада се ради о претрази у дубину, ако је колекција K ред, ради се о претрази у ширину, а ако је у питању ред са приоритетом, ради се о претрази са приоритетом (њу у наставку нећемо детаљно описивати).

Претрага у дубину се често имплементира и рекурзивно.

```
dfs(cvor c):
    ako c nije oznacen:
        oznaci c
        ulazna_obrada_cvora(c)
        za svaku granu cc'
            dfs(c')
        izlazna_obrada_cvora(c)
```

У неким алгоритмима обраду чвора вршимо први пут када на њега наиђемо (тј. је тзв. улазна обрада), док у неким алгоритмима обраду вршимо тек на крају, када обрадимо све потомке чвора.

Претрага у дубину подразумева да се дуж неку путању спуштамо докле год можемо и тек када не можемо да нађемо више непосећених чворова, онда се враћамо назад и разматрамо друге гране из претходно посећених чворова. Претрага у ширину подразумева да се граф обилази по нивоима. Прво се обилази полазни чвор, затим они чворови до којих се из полазног може стићи само путем једне гране, затим они чворови до којих се из полазног може стићи само путем две гране и тако даље.

Задатак: Достижни чворови

За сваки рутер у рачунарској мрежи је познат списак рутера са којима је повезан. Напиши програм који проверава да ли је могуће послати поруку од једног до другог датог рутера.

Улаз: Са стандардног улаза се уноси број рутера n ($1 \leq n \leq 100$), затим број m веза између рутера ($0 \leq m \leq n(n-1)$) и затим у наредних m линија по два различита броја између 1 и n раздвојена размаком која представљају рутере између којих је успостављена веза (први рутер може послати поруку другом). Свака веза се наводи само једном. Након тога се уноси број парова p ($1 \leq p \leq 100$) рутера чију повезаност треба испитати. Сваки пар се описује помоћу два различита броја *start* и *cilj* између 1 и n .

Изаз: За сваки пар рутера који треба испитати, на стандардни излаз исписати да ако је могуће послати поруку од рутера *start* до рутера *cilj* тј. не у супротном.

Пример 1

Улаз	Израз
4	ne
4	da
1 2	
1 3	
3 2	
2 4	
2	
2 3	
1 4	

Пример 2

Улаз	Израз
4	da
4	
1 2	
2 3	
2 4	
3 4	
1	
1 4	

Решење

Повезаност рутера се може описати усмереним графом. Претрагом у дубину или ширину од првог датог рутера обиђемо граф и проверавамо да ли се други рутер налази међу чворовима до којих стижемо током претраге у дубину.

Претрагу у дубину је најједноставније имплементирати рекурзивно.

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

using namespace std;

bool dfs(int r1, int r2, vector<bool>& posecen,
         const vector<vector<int>>& veze) {
    if (r1 == r2)
        return true;
    if (posecen[r1])
        return false;
    posecen[r1] = true;
    for (int r : veze[r1])
        if (dfs(r, r2, posecen, veze))
            return true;
    return false;
}

bool povezani(int r1, int r2, int brojRutera,
              const vector<vector<int>>& veze) {
    vector<bool> posecen(brojRutera, false);
    return dfs(r1, r2, posecen, veze);
}

int main() {
    int brojRutera;
    cin >> brojRutera;
    int brojVeza;
    cin >> brojVeza;
    vector<vector<int>> veze(brojRutera);
    for (int i = 0; i < brojVeza; i++) {
        int a, b;
        cin >> a >> b; a--; b--;
        veze[a].push_back(b);
    }

    int brojParova;
    cin >> brojParova;
    for (int i = 0; i < brojParova; i++) {
        int start, cilj;
        cin >> start >> cilj; start--; cilj--;
```

```
    if (povezani(start, cilj, brojRutera, veze))
        cout << "da" << endl;
    else
        cout << "ne" << endl;
}

return 0;
}
```

Уместо рекурзије, можемо употребити стек.

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

using namespace std;

int main() {
    int brojRutera;
    cin >> brojRutera;
    int brojVeza;
    cin >> brojVeza;
    vector<vector<int>> veze(brojRutera);
    for (int i = 0; i < brojVeza; i++) {
        int a, b;
        cin >> a >> b; a--; b--;
        veze[a].push_back(b);
    }

    int brojParova;
    cin >> brojParova;
    for (int i = 0; i < brojParova; i++) {
        int start, cilj;
        cin >> start >> cilj; start--; cilj--;

        vector<bool> posecen(brojRutera, false);
        stack<int> ruteri;
        ruteri.push(start);
        posecen[start] = true;
        bool povezani = false;
        while (!ruteri.empty() && !povezani) {
            int r = ruteri.top(); ruteri.pop();
            for (auto rr : veze[r]) {
                if (rr == cilj) {
                    povezani = true;
                    break;
                }
                if (!posecen[rr]) {
                    posecen[rr] = true;
                    ruteri.push(rr);
                }
            }
        }
        if (povezani)
            cout << "da" << endl;
        else
            cout << "ne" << endl;
    }
}
```

```
    return 0;
}
```

Задатак: Пећине

Спелеолози се налазе у улазној дворани пећине, на висини тла, чија је надморска висина позната. Пећина има укупно n дворана обележених бројевима од 0 до $n - 1$ (улазна дворана је обележена бројем 0) и до сваке од њих се може стићи коришћењем $n - 1$ ходника који их повезују. Ако се за сваки ходник зна које две дворане повезује и колика је висинска разлика између њих, написати програм који одређује најнижу надморску висину на коју се спелеолози у пећини могу спустити.

Улаз: Са стандардног улаза се читава висина тла (цео број), а затим, из следеће линије природни број n ($1 \leq n \leq 100$). У наредних $n - 1$ линија налазе се по три цела броја раздвојена размацама, која описују ходник: редни број полазне дворане, редни број долазне дворане и висинску разлику између полазне и долазне дворане.

Излаз: На стандардни излаз исписати тражену највећу дубину.

Пример

Улаз	Излаз
278	235
7	
0 1 -20	
0 2 -10	
1 3 -5	
1 4 10	
2 5 -33	
2 6 7	

Решење

Пошто постоји n дворана повезаних са $n - 1$ ходника и пошто се до сваке дворане може стићи, мрежа дворана чини једно дрво и нема циклуса. Висину најниже дворане можемо одредити обиласком дрвета, при чему захваљујући ацикличности не морамо да памтимо да ли смо у некој дворани већ раније били.

Претрагу можемо извршити у дубину и најједноставније је имплементирати је рекурзивно (алтернатива је да употребимо стек).

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

using namespace std;

struct Hodnik {
    int dvoranaDo;
    int razlikaVisina;
    Hodnik(int dvoranaDo_, int razlikaVisina_) {
        dvoranaDo = dvoranaDo_;
        razlikaVisina = razlikaVisina_;
    }
};

int minVisinaDFS(int dvorana, int visina,
                 const vector<vector<Hodnik>>& hodnici) {
    int minVisina = visina;
    for (const Hodnik& h : hodnici[dvorana]) {
        int v = minVisinaDFS(h.dvoranaDo, visina + h.razlikaVisina, hodnici);
        if (v < minVisina)
            minVisina = v;
    }
    return minVisina;
}
```

```
}

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

    int n;
    cin >> n;

    vector<vector<Hodnik>> hodnici(n);
    for (int i = 0; i < n-1; i++) {
        int dvoranaOd, dvoranaDo, razlikaVisina;
        cin >> dvoranaOd >> dvoranaDo >> razlikaVisina;
        hodnici[dvoranaOd].push_back(Hodnik(dvoranaDo, razlikaVisina));
    }

    cout << minVisinaDFS(0, visinaTla, hodnici) << endl;

    return 0;
}
```

Претрагу можемо извршити и у ширину и имплементирати је уз помоћ реда.

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

using namespace std;

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

    int n;
    cin >> n;

    struct Hodnik {
        int dvoranaDo;
        int razlikaVisina;
        Hodnik(int dvoranaDo_, int razlikaVisina_) {
            dvoranaDo = dvoranaDo_;
            razlikaVisina = razlikaVisina_;
        }
    };

    vector<vector<Hodnik>> hodnici(n);
    for (int i = 0; i < n-1; i++) {
        int dvoranaOd, dvoranaDo, razlikaVisina;
        cin >> dvoranaOd >> dvoranaDo >> razlikaVisina;
        hodnici[dvoranaOd].push_back(Hodnik(dvoranaDo, razlikaVisina));
    }

    struct Dvorana {
        int broj, visina;
        Dvorana(int broj_, int visina_) {
            broj = broj_; visina = visina_;
        }
    };
};
```

```

queue<Dvorana> q;
q.push(Dvorana(0, visinaTla));
int min_visina = visinaTla;
while (!q.empty()) {
    Dvorana d = q.front();
    q.pop();
    if (d.visina < min_visina)
        min_visina = d.visina;
    for (const Hodnik& h : hodnici[d.broj])
        q.push(Dvorana(h.dvoranaDo, d.visina + h.razlikaVisina));
}

cout << min_visina << endl;

return 0;
}

```

Задатак: Престолонаследници

У једној земљи краљ је освојио трон и започео своју краљевску лозу. Његових наследника је јако пуно и свако од њих би желео да зна који је по реду да наследи круну. Правило наслеђивања је такво да краља наслеђује прво његов настарији син. Следећи на реду је најстарији унук (ако постоји) и тако редом. Ако неки наследник нема деце, онда је следећи на реду његов најстарији брат, затим његови потомци и тако даље. Напиши програм који одређује редослед наследства одређених краљевих потомака.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$) који представља укупан број особа у краљевом породичном стаблу (укључујући и њега). Након тога се уносе парови облика `roditelj dete` (сваки пар у посебном реду, раздвојен размаком), при чему су деца истог родитеља поређана опадајуће по старости. На крају се уносе имена потомака за које се жели одредити редослед наследства (сваки у посебном реду).

Излаз: За сваког потомка унетог након описа родитељских веза исписати који је по редоследу наследства.

Пример

Улаз	Излаз
19	Harry 6
Elisabeth Charles	Charles 1
Elisabeth Andrew	Charlotte 4
Elisabeth Edward	Louise 12
Elisabeth Anne	James 11
Charles William	Isla 16
Charles Harry	Andrew 7
William George	
William Charlotte	
William Louis	
Anne Peter	
Anne Zara	
Andrew Beatrice	
Andrew Eugenie	
Edward James	
Edward Louise	
Peter Savannah	
Peter Isla	
Zara Mia	
Harry	
Charles	
Charlotte	
Louise	
James	
Isla	
Andrew	

Решење

Услови дати у задатку говоре да је редослед наслеђивања идентичан редоследу обиласка дрвета у дубину (DFS), па задатак решавамо класичном имплементацијом тог алгорита.

Потребно је пре тога решити неколико техничких детаља. Граф тј. дрво ћемо чувати у облику листе суседа. Чуваћемо асоцијативни низ (мапу, речник) у којој се сваком члану краљевске породице придружује низ (листа, вектор) његових наследника, сортираних по приоритету наслеђивања. Пошто у задатку није експлицитно задато ко је монарх, потребно је одредити га из података који се уносе, што се лако може остварити тако што сваком чвору придружимо податак да ли има родитеља или не - монарх ће бити једини ко нема родитеља (те информације можемо реализовати асоцијативним низом имена чланова краљевске породице у истинитосне вредности).

Када су формиране листе суседства и када је одређено ко је монарх, у једном обиласку у дубину обилазимо цело дрво и попуњавамо асоцијативно пресликавање које сваком члану породице додељује његов редни број.

Након што је за свакога одређен редни број, учитавамо једно по једном име и за сваког од њих из асоцијативног пресликавања читамо и на излаз исписујемо редни број.

Обилазак у дубину можемо реализовати тако што на стеку чувамо чворове које треба посетити. На почетку на стек постављамо краља, а затим, све док се стек не испразни, скидамо елемент са врха стека, додељујемо му наредни редни број и његове наследнике додајемо на врх стека (у обратном редоследу). Алтернативно, могли бисмо креирати рекурзивну функцију која прима члана и наредни слободан редни број, додељује тај број члану, а затим се рекурзивно позива за сву његову децу. Функција може да врати наредни слободни редни број (а могуће је редни број чувати и као глобалну променљиву).

```
#include <iostream>
#include <map>
#include <vector>
#include <string>
#include <stack>

using namespace std;

// iterativno implementirana pretraga u dubinu
void dfs(const map<string, vector<string>>& deca, const string& kralj,
        map<string, int>& redosled) {
    // redni broj tokom obilaska
    int r = 0;
    // стек
    stack<string> st;
    // obilazak krecemo od kralja
    st.push(kralj);
    while (!st.empty()) {
        // cvor koji je na vrhu steka je naredni u redosledu nasledstva
        string s = st.top();
        st.pop();
        redosled[s] = r++;
        // ako cvor s ima decu, postavljamo ih na стек u обратном
        // redosledu (da bi se najstariji sin nasao na vrhu steka)
        auto it = deca.find(s);
        if (it != deca.end()) {
            const vector<string>& s_deca = it->second;
            for (auto it = s_deca.rbegin(); it != s_deca.rend(); it++)
                st.push(*it);
        }
    }
}

int main() {
    // da li osoba ima roditelja
```



```

map<string, bool> ima_roditelja;
// ko su sve deca osobe (poredjani po starosti)
map<string, vector<string>> deca;
// broj cvorova
int n;
cin >> n;
// u drvetu postoji n-1 grana
for (int i = 0; i < n - 1; i++) {
    // učitavamo granu drveta
    string roditelj;
    string dete;
    cin >> roditelj >> dete;
    // nasli smo roditelja za učitano dete
    ima_roditelja[dete] = true;
    // ako roditelja učitavamo prvi put, upisujemo da on za sada nema roditelja
    if (ima_roditelja.find(roditelj) == ima_roditelja.end())
        ima_roditelja[roditelj] = false;
    // dodajemo dete u listu dece, na kraj
    deca[roditelj].push_back(dete);
}

// odredjujemo kralja, kao jedinu osobu koja nema roditelja
string kralj;
for (auto it : ima_roditelja)
    if (!it.second)
        kralj = it.first;

// obilaskom u dubinu odredjujemo redosled nasledstva
// svakom imenu pridruzujemo njegov redni broj
map<string, int> redosled;
dfs(deca, kralj, redosled);

// obradjujemo pojedinačne upite
string ime;
while (cin >> ime)
    cout << ime << " " << redosled[ime] << endl;

return 0;
}

```

Претрагу у дубину можемо имплементирати и рекурзивно. Функција прима тренутног родитеља и његов редни број у линији наследства, рекурзивно обрађује све његове потомке (додељујући им кроз рекурзију њихове редне бројеве) и на крају враћа први редни број који је слободан након обраде свих потомака тог родитеља.

```

#include <iostream>
#include <map>
#include <vector>
#include <string>
#include <stack>

using namespace std;

// rekurzivno implementirana pretraga u dubinu funkcija dodeljuje
// roditelju redni broj r, obradjuje sve njegove potomke i vraca
// naredni slobodan redni broj
int dfs(const map<string, vector<string>>& deca, const string& roditelj,
        map<string, int>& redosled, int r) {
    // roditelju pridruzujemo trenutni redni broj

```

```

redosled[roditelj] = r;
// ako roditelj ima decu, postavljamo ih na stek u obratnom
// redosledu (da bi se najstariji sin nasao na vrhu steka)
auto it = deca.find(roditelj);
if (it != deca.end()) {
    const vector<string>& s_deca = it->second;
    for (auto it = s_deca.begin(); it != s_deca.end(); it++) {
        r = dfs(deca, *it, redosled, r+1);
    }
}
return r;
}

// rekurzivno implementirana pretraga u dubinu - funkcija omotac
void dfs(const map<string, vector<string>>& deca, const string& roditelj,
        map<string, int>& redosled) {
    dfs(deca, roditelj, redosled, 0);
}

int main() {
    // da li osoba ima roditelja
    map<string, bool> ima_roditelja;
    // ko su sve deca osobe (poredjani po starosti)
    map<string, vector<string>> deca;
    // broj cvorova
    int n;
    cin >> n;
    // u drvetu postoji n-1 grana
    for (int i = 0; i < n - 1; i++) {
        // ucitavamo granu drveta
        string roditelj;
        string dete;
        cin >> roditelj >> dete;
        // nasli smo roditelja za ucitano dete
        ima_roditelja[dete] = true;
        // ako roditelja ucitavamo prvi put, upisujemo da on za sada nema roditelja
        if (ima_roditelja.find(roditelj) == ima_roditelja.end())
            ima_roditelja[roditelj] = false;
        // dodajemo dete u listu dece, na kraj
        deca[roditelj].push_back(dete);
    }

    // odredjujemo kralja, kao jedinu osobu koja nema roditelja
    string kralj;
    for (auto it : ima_roditelja)
        if (!it.second)
            kralj = it.first;

    // obilaskom u dubinu odredjujemo redosled nasledstva
    // svakom imenu pridruzujemo njegov redni broj
    map<string, int> redosled;
    dfs(deca, kralj, redosled);

    // obradjujemo pojedinačne upite
    string ime;
    while (cin >> ime)
        cout << ime << " " << redosled[ime] << endl;
}

```

```
return 0;
}
```

Задатак: Пећине са тунелима

Спелеолози се налазе у улазној дворани пећине, на висини тла, чија је надморска висина позната. Пећина има укупно n дворана обележених бројевима од 0 до $n - 1$ (улазна дворана је обележена бројем 0) и до сваке од њих се може стићи коришћењем неког од многих тунела који их повезују. Сви тунели су двосмерни. Ако се за сваки тунел зна које две дворане повезује и колика је висинска разлика између њих, написати програм који одређује најнижу надморску висину на коју се спелеолози у пећини могу спустити.

Улаз: Са стандардног улаза се учитава висина тла (цео број), а затим, из следеће линије природни број n ($1 \leq n \leq 100$) који представља број дворана и затим природни број m ($1 \leq m \leq n \cdot (n - 1)/2$) који представља број тунела. У наредних m линија налазе се по три цела броја раздвојена размацама, која описују тунел: редни број полазне дворане, редни број долазне дворане и висинску разлику између полазне и долазне дворане.

Излаз: На стандардни излаз исписати цео број који представља најнижу надморску висину на којој се налази нека дворана.

Пример

Улаз	Излаз
696	679
5	
7	
0 1 -7	
0 4 4	
0 3 -6	
1 2 -10	
1 3 1	
1 4 11	
3 4 10	

Решење

Обиласком графа тунела кренувши од улазне дворане можемо одредити и у низ уписати надморску висину сваке дворане. Након тога тражену најмању висину одређујемо као минимум тог низа. Обилазак можемо имплементирати рекурзивном претрагом у дубину, где у сваком позиву функције прослеђујемо тренутни број дворане и њену надморску висину. Да бисмо избегли циклусе морамо памтити и низ логичких вредности којим се региструју посећене дворане. Приликом поновног уласка у већ посећену дворану одмах прекидамо даљу претрагу.

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

using namespace std;

struct Hodnik {
    int dvoranaDo;
    int razlikaVisina;
    Hodnik(int dvoranaDo_, int razlikaVisina_) {
        dvoranaDo = dvoranaDo_;
        razlikaVisina = razlikaVisina_;
    }
};

void visineDFS(int dvorana, int visina,
               const vector<vector<Hodnik>>& hodnici,
               vector<bool>& posecena,
               vector<int>& visine) {
    if (posecena[dvorana])
```

```

    return;
    posecena[dvorana] = true;
    visine[dvorana] = visina;
    for (const Hodnik& h : hodnici[dvorana]) {
        visineDFS(h.dvoranaDo, visina + h.razlikaVisina,
                  hodnici, posecena, visine);
    }
}

int minVisinaDFS(int dvorana, int visina,
                 const vector<vector<Hodnik>>& hodnici) {
    int n = hodnici.size();
    vector<bool> posecena(n, false);
    vector<int> visine(n);
    visineDFS(dvorana, visina, hodnici, posecena, visine);
    return *min_element(begin(visine), end(visine));
}

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

    int n;
    cin >> n;
    vector<vector<Hodnik>> hodnici(n);

    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        int dvoranaOd, dvoranaDo, razlikaVisina;
        cin >> dvoranaOd >> dvoranaDo >> razlikaVisina;
        hodnici[dvoranaOd].push_back(Hodnik(dvoranaDo, razlikaVisina));
        hodnici[dvoranaDo].push_back(Hodnik(dvoranaOd, -razlikaVisina));
    }

    cout << minVisinaDFS(0, visinaTla, hodnici) << endl;

    return 0;
}

```

Обилазак можемо реализовати и претрагом у ширину (уз коришћење реда).

```

#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <tuple>

using namespace std;

struct Hodnik {
    int dvoranaDo;
    int razlikaVisina;
    Hodnik(int dvoranaDo_, int razlikaVisina_) {
        dvoranaDo = dvoranaDo_;
        razlikaVisina = razlikaVisina_;
    }
};

```

```

int minVisinaBFS(int dvorana, int visina,
                 const vector<vector<Hodnik>>& hodnici) {
    int n = hodnici.size();
    vector<bool> posecena(n, false);
    vector<int> visine(n);
    queue<pair<int, int>> red;
    red.push(make_pair(dvorana, visina));
    while (!red.empty()) {
        tie(dvorana, visina) = red.front();
        red.pop();
        posecena[dvorana] = true;
        visine[dvorana] = visina;
        for (const Hodnik& h : hodnici[dvorana])
            if (!posecena[h.dvoranaDo])
                red.push(make_pair(h.dvoranaDo, visina + h.razlikaVisina));
    }

    return *min_element(begin(visine), end(visine));
}

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

    int n;
    cin >> n;
    vector<vector<Hodnik>> hodnici(n);

    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        int dvoranaOd, dvoranaDo, razlikaVisina;
        cin >> dvoranaOd >> dvoranaDo >> razlikaVisina;
        hodnici[dvoranaOd].push_back(Hodnik(dvoranaDo, razlikaVisina));
        hodnici[dvoranaDo].push_back(Hodnik(dvoranaOd, -razlikaVisina));
    }

    cout << minVisinaBFS(0, visinaTla, hodnici) << endl;

    return 0;
}

```

Задатак: Компоненте повезаности

Напиши програм који одређује компоненте повезаности у неусмереном графу. Компонента повезаности је скуп чворова таквих да из сваког чвора у компоненти постоји пут до сваког другог чвора у компоненти и не постоји пут ни до једног другог чвора ван компоненте.

Улаз: Из прве линије стандардног улаза читава се број чворова ($1 \leq v \leq 100$) и број грана ($0 \leq e \leq \frac{v(v-1)}{2}$) неусмереног графа. У наредних e линија налази се опис грана (пар чворова, раздвојених размаком, при чему се чворови броје од 0 до $v - 1$).

Ишлаз: На стандардни излаз за сваки чвор исписати редни број компоненте којој припада (компоненте се броје од 1). Редослед компоненти одређен је редоследном њихових најмањих елемената (нпр. ако чвор 3 и чвор 7 нису у истој компоненти, редни број компоненте која садржи 3 треба да буде мањи од редног броја компоненте која садржи 7).

Пример

Улаз	Издаз
6 3	1 2 3 2 3 1
1 3	
2 4	
5 0	

Решење

Задатак решавамо тако што редом, из сваког необележеног чвора покрећемо неки алгоритам обиласка графа (на пример, претрагу у дубину) и обележавамо све чворове достижне из полазног чвора. Претрагу у дубину је најлакше реализовати рекурзивно.

Пошто се током обиласка свака грана и сваки чвор посећује само једном, сложеност овог алгоритма је $O(v + e)$.

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

using namespace std;

void dfs(const vector<vector<int>>& A, vector<int>& komponente, int cvor, int k) {
    komponente[cvor] = k;
    for (int sused : A[cvor])
        if (komponente[sused] == 0)
            dfs(A, komponente, sused, k);
}

int main() {
    int v, e;
    cin >> v >> e;
    vector<vector<int>> A(v);
    for (int i = 0; i < e; i++) {
        int a, b;
        cin >> a >> b;
        A[a].push_back(b);
        A[b].push_back(a);
    }

    vector<int> komponente(v, 0);
    int k = 0;
    for (int i = 0; i < v; i++)
        if (komponente[i] == 0)
            dfs(A, komponente, i, ++k);

    for (int i = 0; i < v; i++)
        cout << komponente[i] << " ";
    cout << endl;

    return 0;
}
```

Задатак: Чудна мрежа

У једном рачунарском кабинету потребно је успоставити необичну рачунарску мрежу. Потребно је поставити рачунаре на n столова, при чему на неким столовима могу да стоје обични рачунари (њих имамо на располагању у неограниченом броју), а на неким посебни сервери (њих посебно морамо купити по цени од c_s динара). Неке столове је могуће повезати директно мрежним кабловима, а неке није. Цена успостављања мрежног кабла између било која два стола је c_k динара. Написати програм који одређује најмању цену коју је потребно платити тако да је сваки рачунар или сервер или је мрежним каблом повезан са бар једним сервером (не обавезно директно).

Улаз: Са стандардног улаза се у првом реду уносе бројеви c_s ($0 \leq c_s \leq 1000$) и c_k ($0 \leq c_k \leq 1000$), раздвојени размаком. У наредном реду се уноси број рачунара n ($2 \leq n \leq 50000$) и m број парова рачунара између којих је могуће поставити мрежни кабл $2 \leq m \leq \frac{n(n-1)}{2}$, раздвојени размаком. У наредних m редова уносе се парови бројева између 0 и $n - 1$, раздвојених размаком који одређују рачунаре између којих је могуће поставити кабл.

Издаз: На стандардни издаз исписати тражену најмању цену.

Пример

Улаз	Издаз
850 350	3450
7 6	
0 1	
0 4	
4 2	
1 4	
3 5	
6 5	

Решење

Ако је цена сервера мања или једнака цени једног кабла, тада је најјефтиније на сваки сто поставити по један сервер. Наиме замена сваког сервера обичним рачунаром подразумева повезивање тог рачунара са неким сервером помоћу једног кабла, што повећава укупну цену.

У супротном је оптимално у свакој компоненти повезаности графа поставити по један сервер и све остале столове у тој компоненти попунити обичним рачунарима повезаним са тим сервером. Заиста, ако у некој компоненти не би постојао бар један сервер, онда рачунари у тој компоненти не би могли бити повезани ни са једним сервером, што је супротно условима задатка. Уколико би постојала бар два сервера у некој компоненти, цена не би била оптимална. Наиме један од та два сервера би се могао заменити обичним рачунаром који би се каблом повезао са другим сервером у тој компоненти, чиме би се уместо цене сервера платила цена кабла која је мања.

Број компонената повезаности можемо једноставно одредити обиласком графа (на пример, рекурзивно имплементираним обиласком у дубину), као у задатку **Компоненте повезаности**.

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

using namespace std;

void dfs(const vector<vector<int>>& susedi,
         vector<int>& komponente,
         int cvor, int komponenta) {
    komponente[cvor] = komponenta;
    for (int sused : susedi[cvor])
        if (komponente[sused] == 0)
            dfs(susedi, komponente, sused, komponenta);
}

int broj_kompenenata_povezanosti(const vector<vector<int>>& susedi,
                                int broj_cvorova) {
    vector<int> komponente(broj_cvorova, 0);
    int komponenta = 0;
    for (int cvor = 0; cvor < broj_cvorova; cvor++)
        if (komponente[cvor] == 0)
            dfs(susedi, komponente, cvor, ++komponenta);
    return komponenta;
}

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

```
int broj_racunara, broj_kablova;
long long cena_servera, cena_kabla;
cin >> cena_servera >> cena_kabla;
cin >> broj_racunara >> broj_kablova;
vector<vector<int>> susedi(broj_racunara);
for (int i = 0; i < broj_kablova; i++) {
    int racunar1, racunar2;
    cin >> racunar1 >> racunar2;
    susedi[racunar1].push_back(racunar2);
    susedi[racunar2].push_back(racunar1);
}

if (cena_servera <= cena_kabla)
    cout << broj_racunara * cena_servera << endl;
else {
    int broj_komponentata = broj_komponentata_povezanosti(susedi, broj_racunara);
    int broj_servera = broj_komponentata;
    int broj_obicnih = broj_racunara - broj_servera;
    cout << broj_servera * cena_servera + broj_obicnih * cena_kabla << endl;
}

return 0;
}
```

Задатак: Авионска преседања

Једна авио-компанија заједно са својим партнерима изводи летове између познатих светских аеродрома. Напиши програм који одређује да ли је могуће да се коришћењем тих летова стигне са једног на други дати аеродром и ако јесте, колико је најмање летова потребно.

Улаз: Са стандардног улаза се задаје број m ($1 \leq m \leq 100$) летова које компанија изводи, а затим у наредних m редова опис тих летова (шифра полазног и шифра долазног аеродрома, раздвојени размаком). Након тога се уноси број k ($1 \leq k \leq 100$) путника који су заинтересовани за летове које пружа та компанија, и у наредних k линија описи релација на којима они путују (шифра полазног и шифра долазног аеродрома, раздвојени размаком).

Излаз: За сваког од k путника на стандардни излаз исписати најмањи број летова помоћу којих могу да остваре жељено путовање или реч не ако такво путовање није могуће остварити помоћу летова које компанија изводи.

Пример

Улаз	Излаз
7	2
BEG FRA	не
FRA MUC	3
FRA JFK	
BEG MUC	
MUC LAX	
LAX JFK	
LAX ORD	
3	
BEG JFK	
MUC BEG	
BEG ORD	

Објашњење

Од Београда (BEG) до Њујорка (JFK) може се стићи преко Франкфурта (FRA). Од Минхена (MUC) до Београда (BEG) није могуће организовати путовање. Од Београда (BEG) до Чикага (ORD) могуће је путовати преко Минхена (MUC) и Лос Анђелеса (LAX).

Решење

Најкраће путеве у оријентисаном, нетежинском графу можемо најједноставније пронаћи претрагом у ширину. Њу имплементирамо тако што у ред стављамо чворове у редоследу њиховог растојања од полазног чвора. На почетку стављамо полазни чвор, а затим у сваком кораку скидамо чвор са почетка реда и у ред додајемо његове суседе који раније нису посећени. Уз сваки чвор у ред постављамо и његово растојање од полазног чвора. У тренутку када у ред треба ставити долазни чвор, знамо његово најкраће растојање. Ако се ред испразни пре него што се долазни чвор постави у њега, онда полазни и долазни чвор нису повезани.

```
#include <iostream>
#include <map>
#include <vector>
#include <string>
#include <queue>
#include <set>

using namespace std;

int brojPresedanjaBFS(const string& aerodromOd, const string& aerodromDo,
                     const map<string, vector<string>>& letovi) {
    set<string> posecen;
    queue<pair<string, int>> red;
    red.emplace(aerodromOd, 0);
    while (!red.empty()) {
        string aerodrom;
        int broj;
        tie(aerodrom, broj) = red.front();
        red.pop();
        posecen.insert(aerodrom);

        auto it = letovi.find(aerodrom);
        if (it != letovi.end()) {
            for (const string& s : it->second) {
                if (posecen.find(s) != posecen.end())
                    continue;
                if (s == aerodromDo)
                    return broj+1;
                red.emplace(s, broj+1);
            }
        }
    }
    return -1;
}

int main() {
    int m;
    cin >> m;
    map<string, vector<string>> letovi;
    for (int i = 0; i < m; i++) {
        string aerodromOd, aerodromDo;
        cin >> aerodromOd >> aerodromDo;
        letovi[aerodromOd].push_back(aerodromDo);
    }

    int k;
    cin >> k;
    for (int i = 0; i < k; i++) {
        string aerodromOd, aerodromDo;
        cin >> aerodromOd >> aerodromDo;
```

```
int broj = brojPresedanjaBFS(aerodromOd, aerodromDo, letovi);
if (broj == -1)
    cout << "ne" << endl;
else
    cout << broj << endl;
}
return 0;
}
```

1.3 Тополошко сортирање

Задатак: Редослед послова

Да би се изградио аутомобил, потребно је урадити низ послова. Неки послови зависе од других (на пример, пре него што се уграде точкови, потребно је да се уграде осовине). Напиши програм који одређује могући редослед извршавања ових послова у коме су сва ограничења задовољена.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 50000$), затим број m ($1 \leq m \leq 10n$) и након тога m парова бројева x_i, y_i ($0 \leq x_i, y_i < n$), раздвојених размаком, који означавају да је посао y_i неопходно урадити пре посла x_i .

Излаз: На стандардни излаз исписати n бројева послова у неком редоследу у ком их је могуће извршити (такав редослед ће гарантовано бити могуће направити). Сваки број исписати у посебном реду.

Пример

Улаз	Излаз
6	2
6	5
3 1	0
3 2	1
4 2	3
4 5	4
1 0	
0 5	

Решење

Зависности између послова се могу описати графом у коме постоји грана од чвора x до чвора y ако и само ако посао x зависи од чвора y . Међутим, за неке алгоритме је згодно посматрати и обратан граф у коме постоји града од чвора x до чвора y ако и само ако посао y зависи од чвора x .

Задатак се једноставно може решити тако што се пронађе неки од послова који се могу одрадити, уради се и тако се проблем редукује на проблем истог облика и мање димензије. Ова индуктивно-рекурзивна конструкција се завршава када се не остане више ни један посао који треба урадити. Посао се може урадити ако и само ако не постоје послови од којих он зависи (тј. ако у графу не постоје гране између тог посла и посла од ког он зависи). Када се посао обради, он се избацује (означава се као урађен) заједно са свим гранама између њега и послова који од њега зависе и поступак се наставља, све док се свих послови не ураде. Овај алгоритам је познат под именом *Канов алгоритам*.

Пошто је у сваком кораку Кановог алгоритма потребно одредити послове који зависе од посла који се управо ради, граф ћемо репрезентовати обратом и за сваки чвор x ћемо чувати низ чворова који зависе од посла x . У таквом графу посао се може извршити ако му је улазни степен 0 и одржаваћемо низ улазних степена чворова. Када се неки посао уради, улазни степени свих свих његових суседа се смањују за 1.

У наивној имплементацији претрагу за неким неурђеним послом чији је излазни степен 0 вршимо линеарном претрагом тог низа. Сложеност те линеарне претраге је $O(n)$ и пошто се она спроводи за сваки чвор, сложеност проналажења посла који се може извршити је $O(n^2)$. Сложеност полазног учитавања улазних степена чворова је $O(n + m)$ (гради се низ од n елемената и обрађује се свака од m грана). Такође, улазни степени се ажурирају приликом сваког избацивања гране из графа, па ће ажурирање улазних степена чворова бити сложености $O(m)$. Наравно, потребно је $O(n + m)$ корака да се прочита и складишти граф. Дакле, укупна сложеност овакве имплементације је $O(n^2 + m)$.

```

#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> izlazni_stepen(n);
    vector<vector<int>> prec(n);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        int x, y;
        cin >> x >> y;
        izlazni_stepen[x]++;
        prec[y].push_back(x);
    }

    vector<bool> resen(n, false);
    for (int i = 0; i < n; i++) {
        int cvor;
        for (cvor = 0; cvor < n; cvor++) {
            if (!resen[cvor] && izlazni_stepen[cvor] == 0)
                break;
        }

        if (cvor == n)
            return 1;

        cout << cvor << endl;
        resen[cvor] = true;
        for (int predak : prec[cvor])
            izlazni_stepen[predak]--;
    }

    return 0;
}

```

Имплементација се значајно може убрзати ако све послове за које знамо да су степена 0 и да се могу урадити чувају у радној листи (реду). Након израчунавања почетних улазних степена чворова, у ред стављамо све оне чворове који имају степен 0. Узимамо један по један чвор из реда, умањујемо улазне степене пословима који од њих зависе и ако детектујемо да се неки степен тиме редукује на 0, тај посао одмах додајемо на крај реда. Сваки посао највише једном бива стављен и скинут из реда, па је укупна сложеност идентификације послова који се наредни раде $O(n)$. И у овом случају, потребно је $O(n + m)$ корака да се прочита и складишти граф, као и да се израчунају улазни степени полазног графа, као и $O(m)$ корака да се ажурирају вредности степена чворова током избацивања свих грана. Укупна сложеност је, дакле, $O(n + m)$.

```

#include <iostream>
#include <vector>
#include <queue>

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> stepen(n);
    vector<vector<int>> prec(n);

```

```
int m;
cin >> m;
for (int i = 0; i < m; i++) {
    int x, y;
    cin >> x >> y;
    stepen[x]++;
    preci[y].push_back(x);
}

queue<int> nezavisni_cvorovi;
for (int cvor = 0; cvor < n; cvor++)
    if (stepen[cvor] == 0)
        nezavisni_cvorovi.push(cvor);

while (!nezavisni_cvorovi.empty()) {
    int cvor = nezavisni_cvorovi.front();
    cout << cvor << endl;
    nezavisni_cvorovi.pop();
    for (int predak : preci[cvor])
        if (--stepen[predak] == 0)
            nezavisni_cvorovi.push(predak);
}

return 0;
}
```

Другачији алгоритам може бити заснован на претрази у дубину. Знамо да посао мора бити урађен после свих послова од којих зависи. Зато приликом посете сваком чвору обрађујемо све све његове наследнике (у графу у којој грана од x ка y представља то да посао x зависи од посла y) и на крају серије рекурзивних позива текући чвор додајемо на врх стека. Претрагу позивамо из сваког чвора, и прекидамо је моментално ако установимо да је неки чвор раније био посећен. Након свих позива позива на стеку ће се наћи послови које треба урадити у обрнутом редоследу (посао који треба урадити последњи наћи ће се на врху стека). Сложеност овог приступа је $O(n + m)$.

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

using namespace std;

void dfs(int cvor, vector<bool>& posecen, stack<int>& stek,
        const vector<vector<int>>& susedi) {
    // preskacemo sve ranije posecene cvorove
    if (posecen[cvor])
        return;
    posecen[cvor] = true;
    // pre tekuceg cvora moramo obraditi sve poslove od kojih on zavisi
    for (int sused : susedi[cvor])
        dfs(sused, posecen, stek, susedi);
    // sada mozemo obraditi i tekuci posao
    stek.push(cvor);
}

// topolosko sortiranje
vector<int> top_sort(const vector<vector<int>>& susedi) {
    int n = susedi.size();
    // pokrecemo pretragu u dubinu iz svakog cvora
    vector<bool> posecen(n, false);
    stack<int> stek;
```

```

for (int cvor = 0; cvor < n; cvor++)
    dfs(cvor, posecen, stek, susedi);

// obracemo redosled poslova sa steka
vector<int> redosled(n);
for (int i = n-1; !stek.empty(); i--) {
    redosled[i] = stek.top();
    stek.pop();
}
return redosled;
}

int main() {
    // učitavamo graf
    int n;
    cin >> n;
    int m;
    cin >> m;
    vector<vector<int>> susedi(n);
    for (int i = 0; i < m; i++) {
        int x, y;
        cin >> x >> y;
        susedi[x].push_back(y);
    }

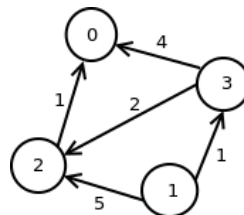
    // odredjujemo i ispisujemo topoloski redosled
    vector<int> redosled = top_sort(susedi);
    for (int cvor : redosled)
        cout << cvor << endl;

    return 0;
}

```

1.4 Најкраћи путеви од једног чвора

У наставку ћемо претпоставити да радимо са тежинским графовима тј. да је свакој грани графа придружена нека тежина (то обично може бити дужина гране тј. растојање између два суседна чвора).



Слика 1.2: Тежински граф

Тежински граф ћемо обично представљати или матрицом повезаности у којој се уместо логичких вредности чувају тежине грана (уз неку специјалну нумеричку вредност која означава да чворови нису повезани) или листама повезаности где се у сваком елементу листе повезаности чува индекс крајњег чвора и тежина гране.

У језику C++ репрезентација графа може бити било матрица облика

```
Tezina A[MAX][MAX];
```

било динамички алоцирана матрица облика

```
vector<vector<Tezina>> A(n);
```

1.4. НАЈКРАЋИ ПУТЕВИ ОД ЈЕДНОГ ЧВОРА

```
for (int i = 0; i < n; i++)  
    A[i].resize(n);
```

Ако користимо листе повезаности, онда можемо употребити структуру података облика

```
vector<vector<pair<Cvor, Tezina>> A(n);
```

која нам омогућава да веома једноставно додајемо нове гране.

```
A[cvor0d].emplace_back(cvor0d, tezina);
```

При том тип *Tezina* означава тежине грана (оне могу бити цели или реални бројеви), док тип *Cvor* означава индексе чворова (и они су по правилу целобројни).

Дајкстрин алгоритам је алгоритам који нам омогућава да ефикасно пронађемо најкраће путеве од једног фиксираних чвора, до свих осталих чворова у графу. Посебно, њиме можемо наћи најкраће растојање између два унапред дата чвора. У наставку ћемо претпоставити да су дужине свих грана у графу ненегативне.

Инваријанта овог алгоритма је то да након k корака извршавања алгоритма уметмо да пронађемо k чворова најближих почетном чвору v и дужине најкраћих путева до њих. У почетку знамо само да је чвор v најближи сам себи и да је дужина пута од v до v једна нули. Означимо са V_k скуп који се састоји од k најближих чворова чвору v , укључујући и v . Проблем је пронаћи чвор w који је најближи чвору v међу чворовима ван V_k , и пронаћи дужину најкраћег пута од v до w . Најкраћи пут од v до w може да садржи само чворове из V_k . Он не може да садржи неки чвор y ван V_k , јер би чвор y био ближи чвору v од w . Према томе, да бисмо пронашли чвор w , довољно је да проверимо гране које спајају чворове из V_k са чворовима који нису у V_k ; све друге гране се за сада могу игнорисати. Нека је (u, z) грана таква да је $u \in V_k$ и $z \notin V_k$. Таква грана одређује пут од v до z који се састоји од најкраћег пута од v до u (који је већ познат) и гране (u, z) . Довољно је упоредити све такве путеве и изабрати најкраћи међу њима. Дакле, чвор w је чвор за који је најмања дужина

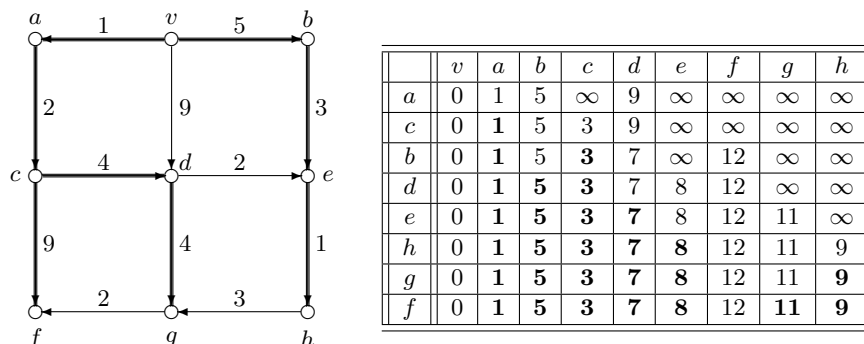
$$\min\{d(v, u) + d(u, w) \mid u \in V_k\}.$$

При том, $d(v, u)$ нам је већ познато минимално растојање од v до u , док је $d(u, w)$ дужина гране између u и w .

У имплементацији можемо у једном низу одржавати текуће процене најкраћих растојања од v до сваког другог чвора. У почетку у низ на место чвора v уписујемо 0, док је дужина свих других путева непозната и у низ уписујемо $+\infty$. Чворови који се налазе у тренутном скупу V_k ћемо на неки начин означавати (на пример, одржавањем помоћног низа у коме логичким вредностима обележавамо означене чворове). У сваком кораку из низа бирамо онај неозначени чвор w који има тренутно најмање растојање и додајемо га у скуп V_k тако што га означавамо. Његово тренутно растојање од чвора v (које је уписано у низу на месту чвора w) је уједно и најкраће растојање од чвора v до свих чворова ван V_k . Зато је w чвор који је $k + 1$ и њиме проширујемо скуп V_k . У том тренутку ажурирамо остала растојања у низу. Нагласимо да приликом проширивања скупа V_k чвором w нема потребе ажурирати путање из чворова који се већ налазе у V_k , већ је довољно ажурирати само путање које воде преко w . То радимо тако што за суседе чвора w проверавамо да ли је збир најкраћег пута од v до w (тренутна вредност у низу на месту чвора w) и дужине гране од w до тог чвора мања од тренутне процене најмање дужине пута дог тог чвора (тј. тренутне вредности у низу на месту тог чвора).

Пример извршавања Дајкстриног алгоритма за налажење најкраћих путева од чвора v у графу дат је на слици. Прва врста односи се само на путеве од једне гране из v . Бира се најкраћи пут, у овом случају он води ка чвору a . Друга врста показује поправке дужина путева укључујући сада све путеве од једне гране из v или a , и најкраћи пут сада води до c . У свакој линији бира се нови чвор, и приказују се дужине тренутних најкраћих путева од v до свих чворова. Подебљана су растојања за која се поуздано зна да су најкраћа (ти чворови су означени).

Пошто је у сваком кораку потребно пронаћи минимум скупа бројева и након тога уклањати минимум из скупа, уместо низа, можемо користити ред са приоритетом у којем чувамо скуп неозначених чворова, уређен на основу тренутне процене растојања од чвора v . За то можемо користити ред са приоритетом (тј. хип). Кључни проблем ове имплементације је то што се након проширивања скупа V_k неке процене дужине пута смањују, па је потребно ажурирати вредности у реду са приоритетом. Редови са приоритетом обично не подржавају директно такве операције. Један начин је да се креира специјализована имплементација која би подржала могућност смањивања вредности у реду, а друга је да се употреби тзв. техника лењог брисања. У том случају се након смањивања процене растојања неког чвора у ред просто додаје нови елемент који



Слика 1.3: Пример примене Дајкстриног алгоритма.

представља смањено растојање. Тиме се за исти чвор у реду истовремено чува више разних процена растојања. У тренутку када се чвор w убацује у V_k из реда ће бити извађено његово најмање растојање од v (ако за чвор w постоје неке раније процене, оне ће сигурно бити веће и још ће се налазити у реду). Ако се касније деси да се из реда извади чвор w на основу неке раније процене, то ћемо лако препознати на основу тога што је w већ раније био убачен у чвор V_k и означен, тако да ћемо тај елемент извађен из реда просто занемарити и прећи ћемо на наредни елемент из реда.

Задатак: Најкраћи пут између два града

Градови су повезани путевима и за сваки пут је познато време потребно да се прође њиме (време потребно да се прође у једном и у другом смеру не мора бити исто). Напиши програм који одређује најкраћи пут између два дата града.

Улаз: Са стандардног улаза се уноси број градова n ($1 \leq n \leq 1000$), у наредном реду број путева m ($1 \leq m \leq n^2 - n$) и затим у наредних m редова описи путева (два цела броја који представљају редне бројеве градова, који се броје од нуле и након тога позитиван реалан број који представља време да се пут пређе). У последња два реда се налазе редни број стартног и редни број циљног града.

Израз: На стандардни излаз исписати минимално време потребно да се стигне од стартног до циљног града, заокружено на пет децимала, а затим, у наредном реду, редне бројеве градова на путу од старта до циља. Ако се од старта до циља не може стићи учитаним путевима, само исписати не.

Пример

Улаз	Израз
4	4.10000
6	0 2 3
0 1 1.2	
1 2 1.3	
0 2 2.0	
0 3 5.7	
1 3 4.6	
2 3 2.1	
0	
3	

Решење

Задатак се решава Дајкстриним алгоритмом. Ако у сваком кораку алгоритма најближи чвор проналазимо обиласка низа тренутних растојања, сложеност алгоритма је $O(n^2)$.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <stack>
#include <limits>
#include <utility>
#include <tuple>
```

1.4. НАЈКРАЋИ ПУТЕВИ ОД ЈЕДНОГ ЧВОРА

```
using namespace std;

typedef int Cvor;
typedef double Duzina;
typedef pair<Duzina, Cvor> Par;

const Duzina INF = numeric_limits<Duzina>::infinity();

int main() {
    ios_base::sync_with_stdio(false);
    // učitavamo graf - koristimo liste suseda
    int n;
    cin >> n;
    vector<vector<Par>> susedi(n);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        Cvor cvorOd, cvorDo; Duzina duzina;
        cin >> cvorOd >> cvorDo >> duzina;
        susedi[cvorOd].emplace_back(duzina, cvorDo);
    }
    // učitavamo startni i ciljni cvor
    int start, cilj;
    cin >> start >> cilj;

    // za svaki cvor cuvamo duzinu najkraceg poznatog puta od startnog cvora
    vector<Duzina> duzinaPuti(n, INF);
    // zatim da li ta procena predstavlja upravo najkraci moguci put
    vector<bool> reseni(n, false);
    // kao i prethodni cvor na tom trenutno procenjenom najkracem putu
    vector<Cvor> roditelji(n, -1);
    // u pocetku jedino znamo rastojanje do startnog cvora
    duzinaPuti[start] = 0.0;
    // dok ne odredimo najkraci put za sve cvorove
    int brojResenih = 0;
    while (brojResenih < n) {
        // odredjujemo nereseni cvor koji je trenutno najblizi startnom
        Cvor cvorMin = 0; Duzina minDuzina = INF;
        for (Cvor cvor = 0; cvor < n; cvor++)
            if (!reseni[cvor] && duzinaPuti[cvor] < minDuzina) {
                cvorMin = cvor;
                minDuzina = duzinaPuti[cvor];
            }

        // nije moguće da postoji bolji put do cvoraMin - belezimo da je on
        // resen i povecavamo broj resenih cvorova
        reseni[cvorMin] = true;
        brojResenih++;
        // ako je za ciljni cvor odredjeno rastojanje, nema potrebe vrsiti
        // dalju pretragu
        if (cvorMin == cilj)
            break;
        // analiziramo susede cvoraMin
        for (const auto& p : susedi[cvorMin]) {
            Cvor cvor; Duzina duzina;
            tie(duzina, cvor) = p;
            // ako je potrebno, azuriramo duzine puta do njegovih suseda
```



```

    if (!resen[cvor] && minDuzina + duzina < duzinaPut[cvor]) {
        duzinaPut[cvor] = minDuzina + duzina;
        roditelji[cvor] = cvorMin;
    }
}
}

// ako put do cilja postoji
if (duzinaPut[cilj] < INF) {
    // ispisujemo duzinu najkraceg puta
    cout << fixed << showpoint << setprecision(5)
        << duzinaPut[cilj] << endl;
    // pratimo put od cilja do starta, unatrag
    // da bismo obrnuli put, koristimo stek
    stack<Cvor> put;
    put.push(cilj);
    while (roditelji[cilj] != -1) {
        cilj = roditelji[cilj];
        put.push(cilj);
    }
    // ispisujemo put od starta do cilja
    while (!put.empty()) {
        cout << put.top() << " ";
        put.pop();
    }
    cout << endl;
} else
    // prijavljujemo da put do cilja ne postoji
    cout << "ne" << endl;

return 0;
}

```

Ефикасност је боља (осим у случају веома густих графова) ако користимо ред са приоритетом. Пошто у библиотечком реду не можемо једноставно ажурирати вредности елемената, користимо веома једноставну технику лењог брисања (додајемо елемент са мањом вредношћу, а стари елемент, са већом вредношћу бришемо из реда тек када исплива као најмањи елемент реда - тада га просто игноришемо, јер знамо да је чвор коме тај елемент одговара већ решен).

Временска сложеност оваког решења је $O((n + m) \log m)$, где је n број чворова, а m број грана у графу. Наиме, ако је циљни чвор најдаљи, тада је потребно извршити бар n операција вађења чворова из реда, за шта нам је потребно бар $O(n \log m)$ операција (јер је вађење минималног чвора операција која логаритамски зависи од броја елемената у реду, а у реду, због лењог брисања може постојати m елемената). Због лењог брисања може бити чак m операција вађења елемената из реда се може поновити m пута, што доноси $O(m \log m)$ операција. Такође, је због лењог брисања могуће да се изврши m уметања у ред што је такође $O(m \log m)$ операција. Дакле, укупан број операција је $O((n + m) \log m)$. Када би се уместо лењог брисања мењале вредности у реду (што тражи ручну имплементацију реда), у реду би увек било највише n елемената, па би сложеност била $O((n + m) \log n)$. Пошто је у графу увек $m = O(n^2)$ (усмерени граф има највише n^2 грана), асимптотска сложеност би била иста као и са лењим брисањем. Без лењог брисања, са специфичним имплементацијама реда са приоритетом (помоћу тзв. Фибоначијевог хипа) могуће је постићи сложеност $O(m + n \log n)$, што није значајно боље. Сложеност је најгора у густом графу $O(n^2) \log n$, што је асимптотски лошије него у случају имплементације без реда са приоритетом. Ако је граф редак тј. ако посматрамо класу графова у којој је $m = O(n)$, тада је сложеност повољнија и износи $O(n \log n)$.

Што се тиче меморијске сложености, са лењим брисањем ред са приоритетом заузима $O(m)$ додатне меморије. С обзиром да је за представљање графа потребно бар $O(n + m)$ меморије, ово не повећава асимптотски меморијску сложеност.

```

#include <iostream>
#include <iomanip>

```

1.4. НАЈКРАЋИ ПУТЕВИ ОД ЈЕДНОГ ЧВОРА

```
#include <vector>
#include <stack>
#include <queue>
#include <limits>
#include <utility>
#include <tuple>

using namespace std;

typedef int Cvor;
typedef double Duzina;
typedef pair<Duzina, Cvor> Par;
typedef tuple<Duzina, Cvor, Cvor> Trojka;

const Duzina INF = numeric_limits<Duzina>::infinity();

int main() {
    ios_base::sync_with_stdio(false);
    // učitavamo graf - koristimo liste suseda
    int n;
    cin >> n;
    vector<vector<Par>> susedi(n);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        Cvor cvorOd, cvorDo; Duzina duzina;
        cin >> cvorOd >> cvorDo >> duzina;
        susedi[cvorOd].emplace_back(duzina, cvorDo);
    }
    // učitavamo startni i ciljni cvor
    int start, cilj;
    cin >> start >> cilj;

    // za svaki cvor cuvamo duzinu najkraceg poznatog puta od startnog cvora
    vector<Duzina> duzinaPuti(n, INF);
    // zatim da li ta procena predstavlja upravo najkraci moguci put
    vector<bool> resen(n, false);
    // kao i prethodni cvor na tom trenutno procenjenom najkracem putu
    vector<Cvor> roditelji(n, -1);
    // u pocetku jedino znamo rastojanje do startnog cvora
    duzinaPuti[start] = 0.0;
    // da bismo brze odredjivali trenutno najblizi cvor, cvorove cuvamo u
    // redu sa prioriteto - za svaki cvor znamo procenu najkraceg puta od
    // starta i prethodni cvor na tom najkracem putu
    priority_queue<Trojka, vector<Trojka>, greater<Trojka>> pq;
    // krećemo od startnog cvora
    pq.emplace(0.0, start, -1);
    // dok ima jos cvorova do kojih se moze stici
    while (!pq.empty()) {
        // vadimo iz reda cvor koji je trenutno najblizi startnom
        Cvor cvorMin, roditelj; Duzina minDuzina;
        tie(minDuzina, cvorMin, roditelj) = pq.top(); pq.pop();
        // zbog lenjog brisanja je moguće da je za taj cvor jos ranije
        // pronadjeno krace rastojanje - u tom slucaju je ovo zastareo
        // podatak i prosto ga ignorisemo
        if (resen[cvorMin])
            continue;
        // nije moguće da postoji bolji put do cvoraMin - belezimo da je on
```

```

// resen, i u niz roditelja pamtimo prethodni cvor na najkracem putu
resen[cvorMin] = true;
roditelji[cvorMin] = roditelj;
// ako je za ciljni cvor odredjeno rastojanje, nema potrebe vrsiti
// dalju pretragu
if (cvorMin == cilj)
    break;
// analiziramo susede cvoraMin
for (const auto& p : susedi[cvorMin]) {
    Cvor cvor; Duzina duzina;
    tie(duzina, cvor) = p;
    // ako je potrebno, azuriramo duzine puta do njegovih suseda
    if (!resen[cvor] && minDuzina + duzina < duzinaPuti[cvor]) {
        duzinaPuti[cvor] = minDuzina + duzina;
        pq.emplace(duzinaPuti[cvor], cvor, cvorMin);
    }
}
}

// ako put do cilja postoji
if (duzinaPuti[cilj] < INF) {
    // ispisujemo duzinu najkraceg puta
    cout << fixed << showpoint << setprecision(5)
         << duzinaPuti[cilj] << endl;
    // pratimo put od cilja do starta, unatrag
    // da bismo obrnuli put, koristimo stek
    stack<Cvor> put;
    put.push(cilj);
    while (roditelji[cilj] != -1) {
        cilj = roditelji[cilj];
        put.push(cilj);
    }
    // ispisujemo put od starta do cilja
    while (!put.empty()) {
        cout << put.top() << " ";
        put.pop();
    }
    cout << endl;
} else
    // prijavljujemo da put do cilja ne postoji
    cout << "ne" << endl;

return 0;
}

```

1.5 Најкраћи путеви између свих парова чворова

Уколико желимо да одредимо најкраће путеве између свих парова чворова можемо покренути Дајкстрин алгоритам из сваког чвора у графу, а можемо употребити и неки од специјализованих алгоритама за проналажење свих најкраћих путева. Један од најпознатијих путева тог типа је Флојд-Варшалов алгоритам. Након k корака алгоритма претпоставићемо да су познате најкраће дужине путева између свих чворова, при чему се на тим путевима као међучворови могу користити само чворови из скупа $V_k = \{0, 1, \dots, k-1\}$. У сваком наредном кораку додајемо чвор k и пошто сада анализирамо путеве између чворова i и j који могу водити преко чворова $V_{k+1} = \{0, 1, \dots, k\}$, упоређујемо дужину пута између та два чвора који не иде преко чвора k већ само преко чворова из V_k (ту дужину већ знамо) и дужину која иде преко чвора k која је једнака збиру дужина најкраћег пута од i до k преко V_k и најкраћег пута од k до j преко V_k (обе ове дужине већ знамо). У случају да се дужине свих најкраћих путева чувају у матрици, Флојд-Варшалов алгоритам се веома једноставно имплементира (помоћу три угнеђена циклуса) и сложеност му је $O(n^3)$, где је n број чворова у

1.5. НАЈКРАЋИ ПУТЕВИ ИЗМЕЋУ СВИХ ПАРОВА ЧВОРОВА

графу.

Флојд-Варшалов алгоритам можемо употребити и за одређивање транзитивног затворења дате релације (тада нас не занимају дужине пута, већ само постојање пута између два чвора).

Задатак: Сви најкраћи путеви у густом графу

На једној територији између свака два града постоји директан пут. Неки путеви су лошег квалитета, па је некад брже стићи од града до града ако се иде неким од околних путева. Ако је познато време да се пређе сваки од директних путева (оно може зависити и од смера у којем се пут прелази), напиши програм који за сваки пар градова одређује колико се времена може уштедети ако се не иде директним путем.

Улаз: Са стандардног улаза се учитава број градова n ($1 \leq n \leq 100$), а након тога квадратна матрица димензије $n \times n$ која садржи времена у секундама потребна да се директним путем стигне од града до града (на дијагонали се налазе нуле, а остала времена су цели бројеви између 60 и 3600).

Излаз: На стандардни излаз исписати квадратну матрицу која садржи уштеде за сваки пар градова.

Пример

Улаз	Излаз
4	0 0 90 0
0 190 300 120	0 0 0 50
180 0 240 350	90 180 0 0
290 430 0 80	0 0 0 0
120 170 90 0	

Објашњење

Од града 1 до града 3 боље је ићи преко града 4 (уместо 300 секунди путује се $120 + 90 = 210$ секунди). И од града 3 до града 1 боље је ићи преко града 4 (уместо 290 секунди путује се $80 + 120 = 200$ секунди). Од града 2 до града 4 најбоље је ићи преко града 1 (уместо 350 секунди путује се $120 + 180 = 300$ секунди). Од града 3 до града 2 најбоље је ићи преко града 4 (уместо 430 секунди путује се $80 + 170 = 250$ секунди).

Решење

Када је мрежа путева густа као у овом задатку, најбољи начин да се нађу најкраћи путеви између свих парова градова је да се употреби Флојд-Варшалов алгоритам, чија је сложеност $O(n^3)$. Пошто се и улазни и излазни подаци чувају у матрици димензије $n \times n$, меморијска сложеност је $O(n^2)$.

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

using namespace std;

// Floyd-Varsalovim algoritmom izracunavamo duzine najkracih puteva
// izmedju svaka dva cvora u grafu
vector<vector<int>>> SviNajkraciPutevi(const vector<vector<int>>& D) {
    int n = D.size();
    vector<vector<int>>> minD = D;
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (minD[i][k] + minD[k][j] < minD[i][j])
                    minD[i][j] = minD[i][k] + minD[k][j];
    }

    return minD;
}

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

    // učitavamo vremena za direktne puteve
```

```

int n;
cin >> n;
vector<vector<int>> D(n);
for (int i = 0; i < n; i++) {
    D[i].resize(n);
    for (int j = 0; j < n; j++)
        cin >> D[i][j];
}

// racunamo najkraca vremena
vector<vector<int>> minD = SviNajkraciPutevi(D);

// ispisujemo rezultat
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        cout << D[i][j] - minD[i][j] << " ";
    cout << endl;
}

return 0;
}

```

Још једно могуће решење је да се употреби индуктивно-рекурзивна конструкција по чворовима графа (градовима). Индуктивна претпоставка је да унемо да одредимо све најкраће путеве у графу од n чворова.

Приликом додавања новог чвора i у граф одређујемо најкраће путеве од сваког од n већ обрађених чворова j до њега. Ти путеви су или директни или се последња деоница у њима остварује тако што се од неког већ обрађеног чвора k стигне до чвора i . Дужина од i до j преко чвора k једнака је збиру дужине најкраћег пута од чвора j до чвора k и дужине гране од чвора k до чвора i . Притом, пошто су све тежине грана позитивне, пут од чвора j до чвора k не садржи чвор i (јер би онда унутар њега био већ садржан краћи пут од j до i). Најкраћи пут од j до k који не укључује i већ знамо на основу индуктивне хипотезе. Дакле, поредимо дужину директне гране од j до i и збирове дужина путева од j до k и гране од k до i за свако могуће k из скупа већ обрађених чворова и минимум тих растојања представља најкраћи пут од j до i .

Дужину најкраћих путева од i до осталих чворова у графу одређујемо аналогно.

На крају, додавање новог чвора i у граф можда скраћује дужину неког пута од раније обрађеног чвора j до раније обрађеног чвора k . Потребно је за све такве парове упоредити дужину тренутно најкраћег пута (који не укључује чвор i) и збир дужина најкраћих путева од j до i и од i до k (које смо у првој фази након додавања чвора i одредили).

Додавање сваког новог чвора захтева $O(n^2)$ корака, па је укупна временска сложеност $O(n^3)$ и показује се да је време извршавања веома слично као и када се користи Флојд-Варшалов алгоритам, али је имплементација компликованија.

```

#include <iostream>
#include <vector>

using namespace std;

// Induktivno rekurzivnim postupkom po broju cvorova izracunavamo
// duzine najkracih puteva izmedju svaka dva cvora u grafu
vector<vector<int>> SviNajkraciPutevi(const vector<vector<int>>& D) {
    int n = D.size();
    vector<vector<int>> minD(n);
    for (int i = 0; i < n; i++)
        minD[i].resize(n);

    // dodajemo jedan po jedan cvor
    for (int i = 0; i < n; i++) {
        minD[i][i] = 0;

```

```
// odredjujemo najkrace puteve od cvora i do svih prethodnih
// cvorova j
for (int j = 0; j < i; j++) {
    // pretpostavljamo da je direktno rastojanje najkrace
    minD[i][j] = D[i][j];
    // proveravamo da li je mozda bolji put od i do j koji vodi preko
    // nekog prethodnog cvora k
    for (int k = 0; k < i; k++)
        if (D[i][k] + minD[k][j] < minD[i][j])
            minD[i][j] = D[i][k] + minD[k][j];
}

// odredjujemo najkrace puteve do cvora i od svih prethodnih
// cvorova j
for (int j = 0; j < i; j++) {
    // pretpostavljamo da je direktno rastojanje najkrace
    minD[j][i] = D[j][i];
    // proveravamo da li je mozda bolji put od cvora j do i koji
    // vodi preko nekog prethodnog cvora k
    for (int k = 0; k < i; k++)
        if (minD[j][k] + D[k][i] < minD[j][i])
            minD[j][i] = minD[j][k] + D[k][i];
}

// popravljamo rastojanja od prethodnih cvorova j do prethodnih
// cvorova k, analizirajuci puteve koji vode preko cvora i
for (int j = 0; j < i; j++)
    for (int k = 0; k < i; k++)
        if (minD[j][i] + minD[i][k] < minD[j][k])
            minD[j][k] = minD[j][i] + minD[i][k];
}

return minD;
}

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

    int n;
    cin >> n;
    vector<vector<int>> D(n);
    for (int i = 0; i < n; i++) {
        D[i].resize(n);
        for (int j = 0; j < n; j++)
            cin >> D[i][j];
    }

    auto minD = SviNajkraciPutevi(D);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << D[i][j] - minD[i][j] << " ";
        cout << endl;
    }

    return 0;
}
```

Једна алтернатива може бити да се из сваког града одреде најкраћа растојања до свих других градова по-

кретањем Дајкстриног алгоритма из тог града. Пошто је граф густ (заправо потпун), имплементација се може направити и елементарно, одређивањем минимума обиласком низа растојања, у линеарној сложености. Сложеност такве имплементације Дајкстриног алгоритма је $O(n^2)$, а пошто се она покреће из сваког чвора, укупна сложеност решења задатка је $O(n^3)$.

```
#include <iostream>
#include <vector>
#include <limits>
#include <queue>

using namespace std;

// +beskonacno
const int INF = numeric_limits<int>::max();

// Dajkstrinim algoritmom koji se pokrece iz svakog cvora izracunavamo
// duzine najkracih puteva izmedju svaka dva cvora u grafu
vector<vector<int>> SviNajkraciPutevi(const vector<vector<int>>& D) {
    // broj cvorova
    int n = D.size();
    // racunamo najkraca vremena
    vector<vector<int>> minD(n);

    // pokrecemo Dajkstrin algoritam za svaki od n cvorova
    for (int cvorI = 0; cvorI < n; cvorI++) {
        // skup gradova do kojih je odredjen najkraci put od cvoraI
        vector<bool> resen(n, false);
        // u pocetku samo znamo da je put do cvora i nula
        minD[cvorI].resize(n, INF);
        minD[cvorI][cvorI] = 0;
        // radimo dok nisu odredjene duzine svih najkracih puteva od I
        for (int brojResenih = 0; brojResenih < n; brojResenih++) {
            // trazimo najkraci put od cvoraI do nekog cvora za koji se jos
            // nije definitivno odredio najkraci put
            int odIDoMin = INF;
            int cvorMin = cvorI;
            for (int cvorJ = 0; cvorJ < n; cvorJ++)
                if (!resen[cvorJ] && minD[cvorI][cvorJ] < odIDoMin) {
                    cvorMin = cvorJ;
                    odIDoMin = minD[cvorI][cvorJ];
                }
            // trenutni put do tog cvora je sigurno najmanji
            resen[cvorMin] = true;
            // azuriramo puteve do ostalih neresenih cvorova posmatrajuci
            // puteve preko trenutnog cvora
            for (int cvorJ = 0; cvorJ < n; cvorJ++)
                if (!resen[cvorJ] &&
                    minD[cvorI][cvorJ] > odIDoMin + D[cvorMin][cvorJ])
                    minD[cvorI][cvorJ] = odIDoMin + D[cvorMin][cvorJ];
        }
    }
    return minD;
}

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

    // ucitavamo vremena za direktne puteve
```

```
int n;
cin >> n;
vector<vector<int>> D(n);
for (int i = 0; i < n; i++) {
    D[i].resize(n);
    for (int j = 0; j < n; j++)
        cin >> D[i][j];
}

// odredjujemo najkrace puteve izmedju svih parova cvorova
vector<vector<int>> minD = SviNajkraciPutevi(D);

// ispisujemo rezultat
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        cout << D[i][j] - minD[i][j] << " ";
    cout << endl;
}

return 0;
}
```

Пошто је граф густ и важи да број грана око половине квадрата броја чворова, имплементација Дајкстриног алгоритма преко реда са приоритетом (са лењим брисањем) само успорава алгоритам. Њена сложеност је $O((n + m) \log m)$, где је m број грана, што је $O((n + n^2) \log n^2)$ тј. $O(n^2 \log n)$, када је граф густ (јер је $m = \Theta(n^2)$), па је укупна сложеност $O(n^3 \log n)$. За складиштење реда са приоритетом користи се додатних $O(m)$ меморије (што не повећава укупну сложеност, јер је за било које складиштење графа потребно бар $O(n + m)$ меморије).

```
#include <iostream>
#include <vector>
#include <limits>
#include <queue>

using namespace std;

// +beskonacno
const int INF = numeric_limits<int>::max();

typedef int Cvor;
typedef int Vreme;
typedef pair<Vreme, Cvor> Par;

// Dajkstrinim algoritmom koji se pokrece iz svakog cvora izracunavamo
// duzine najkracih puteva izmedju svaka dva cvora u grafu
vector<vector<int>> SviNajkraciPutevi(const vector<vector<int>>& D) {
    // broj cvorova
    int n = D.size();
    // matrica u kojoj cuvamo duzine najkracih puteva
    vector<vector<int>> minD(n);

    // pokrecemo Dajkstrin algoritam za svaki od n cvorova
    for (int cvorI = 0; cvorI < n; cvorI++) {
        vector<bool> resen(n, false);
        priority_queue<Par, vector<Par>, greater<Par>> pq;
        // minD[cvorI] sadrzi najkraca rastojanja od cvora I do ostalih cvorova
        // u pocetku ne znamo do kojih sve cvorova se moze stici
        minD[cvorI].resize(n, INF);
        // jedino znamo da je rastojanje od cvora do samog sebe 0
    }
}
```



```

pq.push(make_pair(0, cvorI));
minD[cvorI][cvorI] = 0;

// obradjujemo jedan po jedan cvor, sve dok svi ne budu reseni
while (!pq.empty()) {
    // cvor koji je najblizi cvoruI (ako se izuzmu oni koji su
    // reseni)
    Par p = pq.top();
    pq.pop();
    int cvorMin = p.second;
    int odIDoMin = p.first;
    // ako je cvorMin vec resen, ovo je zaostali podatak (zbog lenjog
    // brisanja), pa ga ignorisemo
    if (resen[cvorMin])
        continue;
    // do cvoraMin ne moze postojati brzi put od cvora I, pa je on
    // postaje resen
    resen[cvorMin] = true;
    // analiziramo susede cvoraMin
    for (int cvorJ = 0; cvorJ < n; cvorJ++)
        // azuriramo vremena do njih ako se do njih putuje preko cvoraMin
        if (!resen[cvorJ] &&
            minD[cvorI][cvorJ] > odIDoMin + D[cvorMin][cvorJ]) {
            minD[cvorI][cvorJ] = odIDoMin + D[cvorMin][cvorJ];
            pq.push(make_pair(minD[cvorI][cvorJ], cvorJ));
        }
    }
}

return minD;
}

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

    // ucitavamo vremena za direktne puteve
    int n;
    cin >> n;
    vector<vector<int>> D(n);
    for (int i = 0; i < n; i++) {
        D[i].resize(n);
        for (int j = 0; j < n; j++)
            cin >> D[i][j];
    }

    // odredjujemo najkrace puteve izmedju svih parova cvorova
    vector<vector<int>> minD = SviNajkraciPutevi(D);

    // ispisujemo rezultat
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << D[i][j] - minD[i][j] << " ";
        cout << endl;
    }

    return 0;
}

```

Задатак: Сви најкраћи путеви у ретком графу

Сваки рутер у једној мрежи повезан је са највише 4 друга рутера. За свака два повезана рутера познато је време потребно да се пошаље порука од првог до другог рутера. Везе су такве да је некада боље слати поруку околo, преко низа рутера, него директно. Напиши програм који одређује колико се највише може скратити време комуникације између нека два рутера ако се порука не шаље директно (у обзир узети само парове рутера који су директно повезани).

Улаз: Са стандардног улаза се учитава број рутера n ($1 \leq n \leq 100$), након тога укупан број веза између рутера m ($n - 1 \leq m \leq 4n$), а након тога у наредних m редова опис сваке везе: свака линија садржи број полазног рутера, број долазног рутера и време потребно да се пошаље порука од полазног до долазног рутера, раздвојене са по једним размаком (бројеви рутера су бројеви од 0 до $n - 1$, а време комуникације је цео број између 1 и 1000).

Израз: На стандардни излаз исписати цео број који представља највеће скраћење комуникације коришћењем посредних рутера.

Пример

Улаз	Израз
6	2
7	
0 1 1	
0 2 4	
0 3 3	
0 5 10	
1 2 2	
3 4 4	
4 5 1	

Објашњење

Најкраћи пут од рутера 0 до рутера 2 се скраћује са 4 на 3 (ако се порука шаље преко рутера 1), док се најкраћи пут од рутера 0 до рутера 5 скраћује са 10 на 8 (ако се порука шаље преко рутера 3 и 4). Зато је највеће скраћење једнако 2.

Решење

Задатак може да се решава на потпуно исте начине као и задатак **Сви најкраћи путеви у густом графу**. Кључна разлика је то што је овај пут граф редак (важи да је m највише $4n$, па је $m = O(n)$). Зато је решење у ком се Дајкстрин алгоритам покреће из сваког чвора ефикасније од алгоритма попут Флојд-Варшаловог који истовремено одређују све најкраће путеве.

Ако се користи ред са приоритетом са лењим брисањем, временска сложеност имплементације Дајкстриног алгоритма је $O((n + m) \log m)$. Пошто је граф редак, укупна временска сложеност је $O(n^2 \log n)$. За складиштење реда са приоритетом се користи $O(m)$ додатне меморије (што не повећава асимптотски меморијску сложеност, јер је за складиштење листа повезаности потребно $O(n + m)$ меморије). Ако одређивање највећег смањења за чворове који су суседни неком чвору вршимо непосредно након што завршимо Дајкстрин алгоритам из тог чвора, тада је довољно да чувамо само низ најкраћих растојања (а не матрицу), за шта је довољно $O(n)$ додатне меморије.

```
#include <iostream>
#include <vector>
#include <limits>
#include <queue>

using namespace std;

const int INF = numeric_limits<int>::max();

typedef int Cvor;
typedef int Vreme;
typedef pair<Vreme, Cvor> Par;
```

```

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

    // broj cvorova
    int n;
    cin >> n;

    // lista suseda svakog cvora
    vector<vector<Par>> susedi(n);

    // učitavamo sve grane
    int k;
    cin >> k;
    for (int i = 0; i < k; i++) {
        Cvor cvorOd, cvorDo;
        Vreme vreme;
        cin >> cvorOd >> cvorDo >> vreme;
        susedi[cvorOd].emplace_back(vreme, cvorDo);
    }

    // najveće skraćanje puta - traženi rezultat
    int maxSkracenje = 0;

    // pokrecemo Dajkstrin algoritam za svaki od n cvorova
    for (Cvor cvorI = 0; cvorI < n; cvorI++) {
        // najmanje vreme od cvora i do svih ostalih cvora
        vector<Vreme> minVremeOdI(n, INF);
        // da li je za dati cvor odredjeno najmanje vreme od cvora i
        vector<bool> resen(n, false);

        // red sa prioritetom kojim odredjujemo cvor za koji nije
        // trenutno zasigurno odredjeno najmanje vreme, a koji je trenutno
        // najblizi cvoru i
        priority_queue<Par, vector<Par>, greater<Par>> pq;
        // cvor i sam sebi poruku salje momentalno
        pq.push(make_pair(0, cvorI));
        minVremeOdI[cvorI] = 0;
        // obradjujemo jedan po jedan cvor, sve dok svi ne budu reseni
        while (!pq.empty()) {
            // cvor do kog poruka najbrze stize od cvora i (ako se izuzmu
            // oni koji su reseni)
            Par p = pq.top();
            pq.pop();
            Cvor cvorMin = p.second;
            Vreme vremeOdIDoMin = p.first;
            // ako je cvorMin vec resen, ovo je zaostali podatak (zbog lenjog
            // brisanja), pa ga ignorisemo
            if (resen[cvorMin])
                continue;
            // do cvoraMin ne moze postojati brzi put od cvora I, pa je on
            // postaje resen
            resen[cvorMin] = true;
            // analiziramo susede cvoraMin
            for (const auto& p : susedi[cvorMin]) {
                // azuriramo vremena do njih ako poruka putuje preko cvoraMin
                Cvor cvorJ = p.second;
                Vreme vremeOdMinDoJ = p.first;
                if (!resen[cvorJ] &&

```

1.5. НАЈКРАЋИ ПУТЕВИ ИЗМЕЂУ СВИХ ПАРОВА ЧВОРОВА

```
        minVremeOdI[cvorJ] > vremeOdIDoMin + vremeOdMinDoJ) {
            minVremeOdI[cvorJ] = vremeOdIDoMin + vremeOdMinDoJ;
            pq.push(make_pair(minVremeOdI[cvorJ], cvorJ));
        }
    }
}

// analiziramo skracenja vremena za sve susede cvora I
for (const auto& p : susedi[cvorI]) {
    Cvor cvorJ = p.second;
    Vreme vremeOdIdoJ = p.first;
    maxSkracenje = max(maxSkracenje, vremeOdIdoJ - minVremeOdI[cvorJ]);
}

cout << maxSkracenje << endl;

return 0;
}
```

Ако се не користи ред са приоритетом, сложеност Дајкстриног алгоритма је $O(n^2)$, па је укупна временска сложеност $O(n^3)$. Меморијска сложеност је асимптотски иста као и у случају имплементације помоћу реда са приоритетом (једино што се не користи помоћни ред).

```
#include <iostream>
#include <vector>
#include <limits>
#include <queue>

using namespace std;

const int INF = numeric_limits<int>::max();

typedef int Cvor;
typedef int Vreme;
typedef pair<Vreme, Cvor> Par;

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

    // broj cvorova
    int n;
    cin >> n;

    // lista suseda svakog cvora
    vector<vector<Par>> susedi(n);

    // učitavamo sve grane
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        int cvorOd, cvorDo, vreme;
        cin >> cvorOd >> cvorDo >> vreme;
        susedi[cvorOd].emplace_back(vreme, cvorDo);
    }

    // najveće skracenje puta - traženi rezultat
    Vreme maxSkracenje = 0;
```

```

// pokrecemo Dajkstrin algoritam za svaki od n cvorova
for (Cvor cvorI = 0; cvorI < n; cvorI++) {
    // najmanje vreme od cvora i do svih ostalih cvora
    vector<Vreme> minVremeOdI(n, INF);
    // da li je za dati cvor odredjeno najmanje vreme od cvora i
    vector<bool> resen(n, false);
    // cvor i sam sebi poruku salje momentalno
    minVremeOdI[cvorI] = 0;
    for (int brojResenih = 0; brojResenih < n; brojResenih++) {
        // trazimo najkraci put od cvora i do nekog cvora za koji se jos
        // nije definitivno odredio najkraci put
        Vreme minVreme = INF;
        Cvor minCvor = cvorI;
        for (Cvor cvorJ = 0; cvorJ < n; cvorJ++)
            if (!resen[cvorJ] && minVremeOdI[cvorJ] < minVreme) {
                minCvor = cvorJ;
                minVreme = minVremeOdI[cvorJ];
            }
        // trenutni put do tog cvora je sigurno najmanji
        resen[minCvor] = true;
        // azuriramo puteve do ostalih neresenih cvorova posmatrajuci
        // puteve preko trenutnog cvora
        for (const auto& p : susedi[minCvor]) {
            Cvor cvorK = p.second;
            Vreme vremeOdMinDoK = p.first;
            if (!resen[cvorK] && minVremeOdI[cvorK] > minVreme + vremeOdMinDoK)
                minVremeOdI[cvorK] = minVreme + vremeOdMinDoK;
        }
    }

    // analiziramo skracenja vremena za sve susede cvora I
    for (const auto& p : susedi[cvorI]) {
        Cvor cvorJ = p.second;
        Vreme vremeOdCvoraIdoJ = p.first;
        maxSkracenje = max(maxSkracenje, vremeOdCvoraIdoJ - minVremeOdI[cvorJ]);
    }
}

cout << maxSkracenje << endl;

return 0;
}

```

Једно могуће решење је да се употреби Флојд-Варшалов алгоритам чија је сложеност $O(n^3)$. Пошто је потребно складиштити матрицу најкраћих путева, меморијска сложеност је $O(n^2)$ (иако се помоћу листа повезаности граф складишти у $O(n + m) = O(n)$ меморије). Једноставности ради, у имплементацији можемо чувати и матрицу директних растојања, чиме се меморијска сложеност не повећава асимптотски.

```

#include <iostream>
#include <vector>
#include <limits>

using namespace std;

const int INF = numeric_limits<int>::max();

vector<vector<int>> sviNajkraciPutevi(const vector<vector<int>>& d) {
    int n = d.size();
    vector<vector<int>> minD = d;

```

```
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (minD[i][k] != INF && minD[k][j] != INF &&
                minD[i][k] + minD[k][j] < minD[i][j])
                minD[i][j] = minD[i][k] + minD[k][j];

return minD;
}

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

    // učitavamo vremena za direktne puteve
    int n;
    cin >> n;
    vector<vector<int>> D(n);
    for (int i = 0; i < n; i++)
        D[i].resize(n, INF);

    int k;
    cin >> k;
    for (int i = 0; i < k; i++) {
        int pocetak, kraj, duzina;
        cin >> pocetak >> kraj >> duzina;
        D[pocetak][kraj] = duzina;
    }

    // racunamo najkraca vremena
    vector<vector<int>> minD = sviNajkraciPutevi(D);

    int m = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (D[i][j] != INF && minD[i][j] != INF)
                m = max(m, D[i][j] - minD[i][j]);

    cout << m << endl;

    return 0;
}
```

Индукција по броју чворова такође доводи до алгоритма временске сложености $O(n^3)$, док је меморијска сложеност условљена чувањем матрице најкраћих путева и износи $O(n^2)$.

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

using namespace std;

const int INF = numeric_limits<int>::max();

// odredjujemo sve najkrace puteve indukcijom po broju cvorova
vector<vector<int>> sviNajkraciPutevi(const vector<vector<int>>& d) {
    int n = d.size();
    vector<vector<int>> minD(n);
    for (int i = 0; i < n; i++)
```

```

    minD[i].resize(n, INF);

    // dodajemo jedan po jedan cvor
    for (int i = 0; i < n; i++) {
        minD[i][i] = 0;

        // odredjujemo najkrace puteve od cvora i do svih prethodnih
        // cvorova j
        for (int j = 0; j < i; j++) {
            // pretpostavljamo da je direktno rastojanje najkrace
            minD[i][j] = d[i][j];
            // proveravamo da li je mozda bolji put od i do j koji vodi preko
            // nekog prethodnog cvora k
            for (int k = 0; k < i; k++)
                if (d[i][k] != INF && minD[k][j] != INF &&
                    d[i][k] + minD[k][j] < minD[i][j])
                    minD[i][j] = d[i][k] + minD[k][j];
        }

        // odredjujemo najkrace puteve do cvora i od svih prethodnih
        // cvorova j
        for (int j = 0; j < i; j++) {
            // pretpostavljamo da je direktno rastojanje najkrace
            minD[j][i] = d[j][i];
            // proveravamo da li je mozda bolji put od cvora j do i koji
            // vodi preko nekog prethodnog cvora k
            for (int k = 0; k < i; k++)
                if (minD[j][k] != INF && d[k][i] != INF &&
                    minD[j][k] + d[k][i] < minD[j][i])
                    minD[j][i] = minD[j][k] + d[k][i];
        }

        // popravljamo rastojanja od prethodnih cvorova j do prethodnih
        // cvorova k, analizirajuci puteve koji vode preko cvora i
        for (int j = 0; j < i; j++)
            for (int k = 0; k < i; k++)
                if (minD[j][i] != INF && minD[i][k] != INF &&
                    minD[j][i] + minD[i][k] < minD[j][k])
                    minD[j][k] = minD[j][i] + minD[i][k];
    }

    return minD;
}

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

    // učitavamo direktne puteve
    int n;
    cin >> n;
    vector<vector<int>> d(n);
    for (int i = 0; i < n; i++)
        d[i].resize(n, INF);

    int k;
    cin >> k;
    for (int i = 0; i < k; i++) {
        int pocetak, kraj, duzina;

```

```
cin >> pocetak >> kraj >> duzina;
d[pocetak][kraj] = duzina;
}

// racunamo najkrace puteve
auto minD = sviNajkraciPutevi(d);

int m = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        if (d[i][j] != INF && minD[i][j] != INF)
            m = max(m, d[i][j] - minD[i][j]);
}
cout << m << endl;

return 0;
}
```

1.6 Минимално повезујуће стабло

Минимално повезујуће (каже се и разапињуће) стабло чини подскуп грана повезаног тежинског графа које повезују све чворове, тако да је укупан збир свих грана најмањи могући. Одређивање минималног повезујућег стабла обично вршимо грамзивим алгоритмима (Примовим или Краскеловим).

Примов алгоритам је веома сличан Дајкстрином. У сваком кораку одржава се скуп чворова V_k и минимално повезујуће стабло тог скупа. Скуп се проширује најкраћом граном која повезује неки чвор графа који припада скупу V_k и неки чвор графа који не припада скупу V_k . Једина разлика у односу на Дајкстрин алгоритам је то што се не разматра растојање неозначених чворова од почетног чвора v , већ растојање од текућег скупа V_k . Приликом додавања неког чвора w скупу V_k , ажурирају се само растојања чворова суседних чвору w (растојања осталих чворова од скупа V_k се не мењају). У имплементацији поново можемо користити било низ у коме чувамо текућа растојања, било ред са приоритетом и лењо брисање.

Краскелов алгоритам сортира све гране графа по дужини, неоппадајући и у сваком кораку у граф додаје најкраћу грану која не затвара циклус у односу на текући скуп грана. Гране не чине дрво, већ шуму. Свако дрво у тој шуми представља неку компоненту повезаности и грана затвара циклус ако и само ако повезује два чвора који се налазе унутар исте компоненте повезаности. За ефикасно представљање компоненти повезаности тј. испитивање да ли су два чвора у истој компоненти, тј. ефикасно спајање две различите компоненте граном можемо користити структуру података за представљање дисјунктних подскупова (енгл. disjoint-set тј. union-find).

Задатак: Уштеда каблова

У једном рачунарском кабинету потребно је поставити мрежу тако да су сви рачунари међусобно повезани, али тако да се употреби што мање кабла. Ако је познато који се рачунари могу повезати кабловима и које је растојање између њих, напиши програм који одређује најмању укупну дужину каблова.

Улаз: Са стандардног улаза се уноси број рачунара n ($1 \leq n \leq 5000$), затим број парова рачунара који се могу повезати каблом m ($n - 1 \leq m \leq n(n - 1)/2$) и у наредних m редова подаци о рачунарима који се могу повезати (у сваком реду се налазе два цела броја која представљају редне бројеве рачунара, при чему се рачунари броје од нуле, и један реалан број који представља растојање између та два рачунара).

Издаз: На стандардни излаз исписати само један реалан број заокружен на једну децималу, који представља најмању укупну дужину каблова.

Пример

Улаз	Израз
9	37.0
13	
0 1 4.0	
0 7 8.0	
1 7 11.0	
1 2 8.0	
7 8 7.0	
7 6 1.0	
2 8 2.0	
8 6 6.0	
2 3 7.0	
2 5 4.0	
6 5 2.0	
3 4 9.0	
5 4 10.0	

Решење

Задатак решавамо коришћењем неког од грамзивих алгоритама за конструкцију минималног разапињућег стабла.

Примов алгоритам

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <utility>
#include <queue>
#include <tuple>
#include <limits>

using namespace std;

typedef int Cvor;
typedef double Duzina;
typedef pair<Duzina, Cvor> Par;

// +beskonacno
const Duzina INF = numeric_limits<Duzina>::infinity();

int main() {
    // učitavamo podatke
    ios_base::sync_with_stdio(false);
    int n;
    cin >> n;
    vector<vector<Par>> susedi(n);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        int cvor1, cvor2;
        double duzina;
        cin >> cvor1 >> cvor2 >> duzina;
        susedi[cvor1].emplace_back(duzina, cvor2);
        susedi[cvor2].emplace_back(duzina, cvor1);
    }

    // ukupna duzina kablova u trenutnom stablu
    double ukupnaDuzina = 0.0;
    // najmanje rastojanje svakog cvora od trenutnog stabla
```

```
vector<Duzina> rastojanje(n, INF);
// da li je cvor ukljucen u trenutno stablo
vector<bool> ukljucen(n, false);
// krecemo od praznog stabla - postavljamo rastojanje do cvora 0 na 0.0
// da bi taj cvor prvi bio ukljucen u stablo
rastojanje[0] = 0.0;

// broj cvorova u stablu
int cvorovaUStablu = 0;
// sve dok ne dodamo n cvorova u stablo
while (cvorovaUStablu < n) {
    // pronalazimo cvor koji je najblizi trenutnom stablu
    Cvor cvorMin = 0;
    Duzina minRastojanje = INF; // rastojanje najblizeg cvora od stabla
    for (Cvor cvor = 0; cvor < n; cvor++)
        if (!ukljucen[cvor] && rastojanje[cvor] < minRastojanje) {
            cvorMin = cvor;
            minRastojanje = rastojanje[cvor];
        }
    // ukljucujemo cvorMin u stablo
    ukljucen[cvorMin] = true;
    cvorovaUStablu++;
    // uracunavamo duzinu najkrace grane koja ga spaja sa trenutnim stablom
    ukupnaDuzina += minRastojanje;
    // njegovo rastojanje do stabla je sada 0
    rastojanje[cvorMin] = 0.0;
    // razmatramo sve susede cvoraMin
    for (const Par& p : susedi[cvorMin])
        // azuriramo njihovo rastojanje od trenutnog stabla, ako je to potrebno
        if (p.first < rastojanje[p.second])
            rastojanje[p.second] = p.first;
}

// ispisujemo konacan rezultat
cout << fixed << showpoint << setprecision(1) << ukupnaDuzina << endl;

return 0;
}
```

Примов алгоритам са редом са приоритетом

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <utility>
#include <queue>
#include <tuple>
#include <limits>

using namespace std;

typedef int Cvor;
typedef double Duzina;
typedef pair<Duzina, Cvor> Par;

const Duzina INF = numeric_limits<Duzina>::infinity();

int main() {
    // učitavamo podatke
```

```

ios_base::sync_with_stdio(false);
int n;
cin >> n;
vector<vector<Par>> susedi(n);
int m;
cin >> m;
for (int i = 0; i < m; i++) {
    int cvor1, cvor2;
    double duzina;
    cin >> cvor1 >> cvor2 >> duzina;
    susedi[cvor1].emplace_back(duzina, cvor2);
    susedi[cvor2].emplace_back(duzina, cvor1);
}

// ukupna duzina kablova u trenutnom stablu
double ukupnaDuzina = 0.0;
// najmanje rastojanje svakog cvora od trenutnog stabla
vector<Duzina> rastojanje(n, INF);
// da li je cvor ukljucen u trenutno stablo
vector<bool> ukljucen(n, false);
// red sa prioriteto koji nam pomaze da efikasnije nadjemo cvor koji
// je najblizi trenutnom stablu
priority_queue<Par, vector<Par>, greater<Par>> pq;
// krecemo od praznog stabla - postavljamo rastojanje do cvora 0 na 0.0
// da bi taj cvor prvi bio ukljucen u stablo
rastojanje[0] = 0.0;
pq.emplace(0.0, 0);

// broj cvorova u stablu
int cvorovaUStablu = 0;
// sve dok ne dodamo n cvorova u stablo
while (cvorovaUStablu < n) {
    // pronalazimo cvor koji je najblizi trenutnom stablu koriscenjem
    // reda i uklanjamo ga iz reda
    double minRastojanje; Cvor cvorMin;
    tie(minRastojanje, cvorMin) = pq.top();
    pq.pop();
    // zbog lenjog brisanja, moguće je da je on ranije ukljucen u
    // stablo, pa ga u tom slucaju zanemarujemo
    if (ukljucen[cvorMin])
        continue;
    // ukljucujemo cvorMin u stablo
    ukljucen[cvorMin] = true;
    cvorovaUStablu++;
    // uracunavamo duzinu najkrace grane koja ga spaja sa trenutnim
    // stablom
    ukupnaDuzina += minRastojanje;
    // njegovo rastojanje do stabla je sada 0
    rastojanje[cvorMin] = 0.0;
    // razmatramo sve susede cvoraMin
    for (const Par& p : susedi[cvorMin])
        // azuriramo njihovo rastojanje od trenutnog stabla, ako je to
        // potrebno
        if (p.first < rastojanje[p.second]) {
            rastojanje[p.second] = p.first;
            // ako se rastojanje smanjilo, dodajemo ga u red, pri cemu
            // staru vrednost iz reda (ako postoji) ne brisemo odmah
            pq.push(p);
        }
    }

```

```
    }  
}  
// ispisujemo konacan rezultat  
cout << fixed << showpoint << setprecision(1) << ukupnaDuzina << endl;  
  
return 0;  
}
```

Краскелов алгоритам

```
#include <iostream>  
#include <iomanip>  
#include <vector>  
#include <tuple>  
#include <algorithm>  
#include <numeric>  
  
using namespace std;  
  
// tip grane grafa - duzina ide prva, zbog sortiranja  
typedef int Cvor;  
typedef double Duzina;  
typedef tuple<Duzina, Cvor, Cvor> Grana;  
  
int main() {  
    // učitavamo podatke  
    ios_base::sync_with_stdio(false);  
    int n;  
    cin >> n;  
    int m;  
    cin >> m;  
    vector<Grana> grane(m);  
    for (int i = 0; i < m; i++) {  
        int cvorOd, cvorDo;  
        double duzina;  
        cin >> cvorOd >> cvorDo >> duzina;  
        grane[i] = make_tuple(duzina, cvorOd, cvorDo);  
    }  
    // sortiramo grane u rastucemo redosledu duzina  
    sort(begin(grane), end(grane));  
    // struktura podataka za predstavljanje formiranih grupa  
    // za svaki element pamtimo u kojoj se grupi nalazi  
    vector<int> grupa(n);  
    iota(begin(grupa), end(grupa), 0);  
  
    // ukupna duzina kablova  
    double ukupnaDuzina = 0.0;  
    // broj trenutno dodatih grana  
    int dodatoGrana = 0;  
    for (int i = 0; i < m && dodatoGrana < n-1; i++) {  
        int c1 = get<1>(grane[i]);  
        int c2 = get<2>(grane[i]);  
        // ako trenutna grana spaja cvorove u dve razlicite komponente  
        if (grupa[c1] != grupa[c2]) {  
            int g1 = grupa[c1];  
            // tada spajamo komponente  
            for (int j = 0; j < n; j++)  
                if (grupa[j] == g1)  
                    grupa[j] = grupa[c2];  
            dodatoGrana++;  
        }  
    }  
    cout << fixed << showpoint << setprecision(1) << ukupnaDuzina << endl;  
    return 0;  
}
```

```

    // dodajemo granu u drvo
    Duzina duzina = get<0>(grane[i]);
    ukupnaDuzina += duzina;
    dodatoGrana++;
}
}
// ispisujemo konacan rezultat
cout << fixed << showpoint << setprecision(2) << ukupnaDuzina << endl;
return 0;
}

```

Краскелов алгоритам са структуром за представљање дисјунктних скупова (енгл. union-find).

```

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

using namespace std;

// struktura podataka za predstavljanje disjunktних podskupova (union-find)
struct UF_Cvor {
    int roditelj;
    int rang; // bila bi visina, da se ne vrsi dodatno kompresija staze
};

// inicijalizacija - svaki cvor je u posebnoj komponenti
void UF_Init(vector<UF_Cvor>& uf) {
    for (size_t i = 0; i < uf.size(); i++) {
        uf[i].roditelj = i;
        uf[i].rang = 0;
    }
}

// pronalazenje predstavnika cvora
int UF_Find(vector<UF_Cvor>& uf, int i) {
    while (uf[i].roditelj != i) {
        // kompresija staze - svaki drugi cvor ukazuje na svog dedu
        uf[i].roditelj = uf[uf[i].roditelj].roditelj;
        i = uf[i].roditelj;
    }
    return i;
}

// unija dve komponente
bool UF_Union(vector<UF_Cvor>& uf, int v1, int v2) {
    int v1koren = UF_Find(uf, v1);
    int v2koren = UF_Find(uf, v2);
    // cvorovi v1 i v2 su u istoj komponentni
    if (v1koren == v2koren)
        return false;
    // granu vucemo od grupe sa manjim rangom ka grupi sa vecim
    if (uf[v1koren].rang < uf[v2koren].rang) {
        uf[v1koren].roditelj = v2koren;
    } else if (uf[v1koren].rang > uf[v2koren].rang) {
        uf[v2koren].roditelj = v1koren;
    } else {

```

```
// rangovi su jednaki, pa se rang (visina) novog drveta povecava
uf[v2koren].roditelj = v1koren;
uf[v1koren].rang++;
}
return true;
}

// tip grane grafa - duzina ide prva, zbog sortiranja
typedef int Cvor;
typedef double Duzina;
typedef tuple<Duzina, Cvor, Cvor> Grana;

int main() {
    // učitavamo podatke
    ios_base::sync_with_stdio(false);
    int n;
    cin >> n;
    int m;
    cin >> m;
    vector<Grana> grane(m);
    for (int i = 0; i < m; i++) {
        int cvorOd, cvorDo;
        double duzina;
        cin >> cvorOd >> cvorDo >> duzina;
        grane[i] = make_tuple(duzina, cvorOd, cvorDo);
    }
    // sortiramo grane u rastucemo redosledu duzina
    sort(begin(grane), end(grane));
    // struktura podataka za predstavljanje formiranih grupa
    vector<UF_Cvor> uf(n);
    UF_Init(uf);
    // ukupna duzina kablova
    double ukupnaDuzina = 0.0;
    // broj trenutno dodatih grana
    int dodatoGrana = 0;
    for (int i = 0; i < m && dodatoGrana < n-1; i++) {
        // ako trenutna grana spaja cvorove u dve razlicite komponente,
        // tada spajamo komponente
        if (UF_Union(uf, get<1>(grane[i]), get<2>(grane[i]))) {
            // dodajemo granu u drvo
            ukupnaDuzina += get<0>(grane[i]);
            dodatoGrana++;
        }
    }
    // ispisujemo konacan rezultat
    cout << fixed << showpoint << setprecision(1) << ukupnaDuzina << endl;

    return 0;
}
```

Задатак: Кластери

У једној земљи се организује шампионат у фудбалу. Утакмице ће се играти у n градова, који су повезани путевима. На турниру игра k екипа и организатори желе да поделе те градове у k ($k \leq n$) група, тако да су те групе што више раздвојене, тј. тако да је најкраћи пут између било које две групе градова што је већи могући (да би навијачи били што више раздвојени).

Улаз: Са стандардног улаза учитава се број k који представља тражени број група градова, затим n који представља број градова, затим, затим број p који представља број путева и затим у наредних p редова опис

путева (индекси, кренувши од нуле, два града које пут спаја и затим дужина пута у километрима).

Излаз: На стандардни излаз исписати удаљеност две најближе формиране групе градова.

Пример

Улаз	Излаз
3	10
7	
9	
0 1 10	
1 5 1	
1 3 1	
5 3 1	
4 2 1	
2 6 1	
2 5 12	
1 4 11	
0 6 15	

Објашњење

У првој групи налази се само град 0, у другој градови 1, 3, 5, а у трећој 2, 4, 6. Удаљеност између прве и друге групе је 10, а између прве и треће је 15, а између друге и треће је 11. Намање растојање између две групе градова је 10.

Решење

Кренимо од ситуације у којој је сваки град група за себе. Ако је број градова једнак траженом броју група, то је једина могућа конфигурација, док је у супротном потребно извршити груписање нека два града у исту групу. Најбоље је да се групишу градови који су што мање међусобно удаљени, јер би се у случају груписања нека два даља града смањило најмање растојање између две различите групе. Сличан поступак се наставља и даље, све док се не добије тражени број група. Када је креирано m група, где је $m > k$, потребно је спојити неке две групе. Поново бирамо најкраћу грану која повезује неке две различите групе и те две групе спајамо. Дакле, поступак решавамо грамзивим алгоритмом, који је веома сличан Краскеловом алгоритму за израду минималног повезујућег дрвета. Најмање растојање група биће $k - 1$ -ва по дужини грана која припада том дрвету. Имплементација тече тако што сортирамо све гране по дужини, обилазимо их у растућем редоследу дужине и додајемо једну по једну грану која спаја чворове у различитим компонентама. Чување компонената вршимо коришћењем структуре података за репрезентовање дисјунктних скупова (енгл. union-find). Када се број компонената смањи испод жељеног броја k , последња додата грана представља најмање растојање између неке две од k компонената.

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

using namespace std;

// структура података за predstavljanje disjunktних podskupova (union-find)
struct UF_Cvor {
    int roditelj;
    int rang; // bila bi visina, da se ne vrsi dodatno kompresija staze
};

// inicijalizacija - svaki cvor je u posebnoj komponenti
void UF_Init(vector<UF_Cvor>& uf) {
    for (size_t i = 0; i < uf.size(); i++) {
        uf[i].roditelj = i;
        uf[i].rang = 0;
    }
}
```

```
// pronalazenje predstavnika cvora
int UF_Find(vector<UF_Cvor>& uf, int i) {
    while (uf[i].roditelj != i) {
        // kompresija staze - svaki drugi cvor ukazuje na svog dedu
        uf[i].roditelj = uf[uf[i].roditelj].roditelj;
        i = uf[i].roditelj;
    }
    return i;
}

// unija dve komponente
bool UF_Union(vector<UF_Cvor>& uf, int v1, int v2) {
    int v1koren = UF_Find(uf, v1);
    int v2koren = UF_Find(uf, v2);
    // cvorovi v1 i v2 su u istoj komponentni
    if (v1koren == v2koren)
        return false;
    // granu vucemo od grupe sa manjim rangom ka grupi sa vecim
    if (uf[v1koren].rang < uf[v2].rang) {
        uf[v1koren].roditelj = v2koren;
    } else if (uf[v1koren].rang > uf[v2koren].rang) {
        uf[v2koren].roditelj = v1koren;
    } else {
        // rangovi su jednaki, pa se rang (visina) novog drвета povećava
        uf[v2koren].roditelj = v1koren;
        uf[v1koren].rang++;
    }
    return true;
}

int main() {
    ios_base::sync_with_stdio(false);
    // broj zeljenih grupa
    int k;
    cin >> k;
    // broj gradova i broj puteva
    int n, p;
    cin >> n >> p;
    // učitavamo sve puteve (uredjene trojke)
    typedef tuple<int, int, int> Put;
    vector<Put> putevi(p);
    for (int i = 0; i < p; i++) {
        int g1, g2, duzina;
        cin >> g1 >> g2 >> duzina;
        putevi[i] = make_tuple(duzina, g1, g2);
    }

    // sortiramo sve puteve po duzini
    sort(begin(putevi), end(putevi));

    // struktura podataka za predstavljanje formiranih grupa
    vector<UF_Cvor> uf(n);
    UF_Init(uf);
    // na pocetku je svaki grad u svojoj grupi
    int brojGrupa = n;
    // obilazimo puteve u rastucem redosledu duzine
    for (auto put: putevi) {
        // ako je ovaj put izmedju dve razlicite grupe
```



```
if (UF_Union(uf, get<1>(put), get<2>(put))) {  
    // spajamo dve grupe kojima pripadaju krajevi tog puta  
    brojGrupa--;  
    if (brojGrupa < k) {  
        // ako se broj grupa smanjio ispod k, upravo dodati put je bio  
        // najkraci put izmedju neke dve od k prethodno formiranih grupa  
        cout << get<0>(put) << endl;  
        break;  
    }  
}  
}  
}  
  
return 0;  
}
```

Глава 2

Алгебарски алгоритми

Када извршавамо неку алгебарску операцију, као што је рецимо множење два броја или њихово степеновање, ми у ствари извршавамо неки алгоритам. Ми те операције користимо као градивне елементе у развијању сложенијих алгоритама и често не залазимо дубље у анализу њихове сложености. Међутим, и сами алгоритми сабирања, одузимања, множења и дељења бројева (посебно ако су бројеви дати низовима својих цифара) представљају важне алгебарске алгоритме. У алгебарске алгоритме спадају и многи алгоритми са којима смо се раније сусретали као што су израчунавање вредности броја на основу датих цифара или, насупротив томе, одређивање цифара броја на основу његове вредности, затим разни алгоритми над полиномима као што су израчунавање вредности полинома и множење полинома. У наставку ћемо се бавити алгебарским алгоритмима са којима се до сада нисмо сусретали. Многи од њих играју важну улогу у области криптографије, али и у другим областима.

2.1 Модуларна аритметика

Кажемо да је број r остатак при дељењу броја x бројем y и пишемо $x \bmod y = r$ ако и само ако постоји број q такав да је $x = q \cdot y + r$ и $0 \leq r < y$. Поред тога што са $\bmod$ означавамо бинарну операцију, $\bmod$ се може користити и као ознака бинарне релације у скупу целих бројева. Наиме, писаћемо $a \equiv b \bmod m$ ако $m \mid (a - b)$. Ово је еквивалентно томе да a и b дају исти остатак при дељењу са m . На пример, пишемо $12 \equiv 2 \bmod 5$ јер $5 \mid (12 - 2)$.

Релацију $\bmod$ користимо у неким свакодневним ситуацијама, а да тога често нисмо ни свесни.

- Један од таквих примера је рад са временом. Наиме, за 15 сати након 11 часова рећи ћемо да је 2 сата (што одговара томе да је $11 + 15 \equiv 2 \bmod 24$).
- Слично важи и за дане у недељи, које рачунамо по модулу 7. Ако је данас, рецимо, четвртак и интересује нас који ће дан бити за сто дана, не морамо да бројимо један по један дан. Сваком дану можемо придружити неки остатак при дељењу са 7 (0 - понедељак, 1 - уторак, 2 - среда, 3 - четвртак, 4 - петак, 5 - субота, 6 - недеља), а затим израчунати $(3 + 100) \bmod 7 = 5$ и знаћемо да је за 100 дана субота (проћи ће 14 пуних недеља тј. 98 дана и биће поново четвртак, након 99 дана ће бити петак, а након 100 субота). Аналогно се рачуна и месец или редни број недеље у години.
- Слично важи и приликом рада са угловима (које рачунамо по модулу 360 степени).

Модуларна аритметика има пуно практичних примена: користи се за израчунавање контролних сума за међународне стандардне идентификаторе књига (ISBN бројеви) и идентификаторе банке (IBAN). Модуларна аритметика је и у основи савремених криптографских система.

Бројеви се у рачунарима представљају по модулу 2^k . На пример, у језику C++ бројеви типа `unsigned int` представљају се по модулу 2^{32} . На пример, у језику C# бројеви типа `uint` представљају се по модулу 2^{32} . У наредном коду резултат квадрирања броја 123456789 биће вредност $123456789^2 \bmod 32 = 2537071545$.

```
unsigned int x = 123456789;  
cout << x*x << endl;
```

2.1.1 Сабирање и множење по модулу

Уколико је потребно одредити вредност збира или производа бројева *по модулу* m од помоћи нам могу бити наредне релације:

$$(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$$

$$(a \cdot b) \bmod m = (a \bmod m \cdot b \bmod m) \bmod m$$

Докажимо другу релацију (прва је једноставнија за доказивање, па је остављамо за вежбу). Претпоставимо да је $a = q_a \cdot m + r_a$ и $b = q_b \cdot m + r_b$ за $0 \leq r_a, r_b < m$. Тада важи да је $a \cdot b = (q_a \cdot m + r_a) \cdot (q_b \cdot m + r_b) = (q_a \cdot q_b \cdot m + q_a \cdot r_b + r_a \cdot q_b)m + r_a \cdot r_b$. Ако важи да је $r_a \cdot r_b = q \cdot m + r$ за $0 \leq r < m$, тада је са $a \cdot b = (q_a \cdot q_b \cdot m + q_a \cdot r_b + r_a \cdot q_b + q)m + r$, па је $(a \cdot b) \bmod m = r$. Са друге стране, важи да је $(a \bmod m \cdot b \bmod m) \bmod m = (r_a \cdot r_b) \bmod m = r$, чиме је тврђење доказано.

2.1.2 Одузимање по модулу

Размотримо проблем одређивања разлике бројева b и a по модулу m . На пример, потребно је одредити вредност $(b - a) \bmod m$. Желели бисмо да као вредност овог израза добијемо ненегативну вредност чак и у случају када је $b < a$. Међутим, не постоји сагласност између различитих програмских језика у рачунању вредности $x \% m$ када је x негативан. Наиме, у језику C++ и у језику C# бисмо за вредност остатка добили негативан број: вредност израза $(2 - 7) \% 3$ била би -2 , док би у језику Python као резултат добили позитиван број 1 . Уместо да вршимо испитивање да ли је дељеник негативан, ако је $0 \leq a, b < m$, позитиван резултат је могуће добити израчунавањем вредности израза $(b - a + m) \bmod m$. Додавањем вредности m , вредност $b - a + m$ ће постати сигурно ненегативна (јер разлика $b - a$ не може бити мања од $-m$), а тражење остатка ће практично поништити првобитно додавање вредности m). У претходном смо се ослонили на чињеницу да су и a и b бројеви који су већи или једнаки од 0 и строго мањи од m . Проналажење вредности разлике по модулу m могуће је и у општем случају и важи

$$(B - A) \bmod m = (B \bmod m - A \bmod m + m) \bmod m$$

Докажимо ово тврђење. Подсетимо се да је $x \bmod y = r$ ако и само ако постоји q такав да је $x = q \cdot y + r$ и ако је $0 \leq r < y$. Нека је $A = q_a \cdot m + a$ и $B = q_b \cdot m + b$, за $0 \leq a, b < m$. Зато је $A \bmod m = a$ и $B \bmod m = b$. Нека је $B - A + m = p \cdot m + r$ за неко $0 \leq r < m$. Зато је $(B \bmod m - A \bmod m + m) \bmod m = (b - a + m) \bmod m = r$. Такође, важи и да је $B - A = (q_b - q_a) \cdot m + (b - a) = (q_b - q_a - 1) \cdot m + (b - a + m) = (q_b - q_a - 1 + p) \cdot m + r$, па је и $(B - A) \bmod m = r$, чиме је тврђење доказано.

2.1.3 Степеновање по модулу

Као последица множења по модулу, важи и наредно тврђење, које омогућава да се израчуна степен по модулу:

$$a^n \bmod m = (a \bmod m)^n \bmod m$$

Задатак: Операције по модулу

Напиши програм који одређује последње три цифре збира и последње три цифре производа четири унета цела броја.

Улаз: У сваком реду стандардног улаза уноси се по један цео број из интервала $[1..999]$.

Издаз: На стандардни издаз се исписују број одређен са последње три цифре збира и број одређен са последње три цифре производа унетих бројева (евентуалне водеће нуле се не морају исписати).

Пример

Улаз	Издаз
999	996
999	1
999	
999	

Решење

Идеја која природно прва падне напамет је да се израчунају збир и производ унетих бројева, а онда да се последње три цифре одреде израчунавањем остатка при целобројном дељењу са 1000. Када се у обзир узме распон бројева који се уносе, такво решење би давало коректне резултате у случају збира, међутим, производ бројева може бити много већи од самих бројева и може прекорачити распон типа `int` који користимо за репрезентовање целих бројева и зато резултат може бити погрешан.

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    int a, b, c, d;
    cin >> a >> b >> c >> d;

    cout << (a + b + c + d) % 1000 << endl;
    cout << (a * b * c * d) % 1000 << endl;

    return 0;
}
```

Зато је за израчунавање последње три цифре производа потребно применити мало напреднији алгоритам који користи чињеницу да су за последње три цифре производа релевантне само последње три цифре сваког од чинилаца. Зато је пре множења могуће одредити остатак при дељењу са 1000 сваког од чинилаца, израчунати производ, а онда одредити његове три последње цифре израчунавањем остатка његовог при дељењу са 1000. Аналогну тактику могуће је применити и на израчунавање збира.

Функције за множење и сабирање по модулу засноваћемо на релацијама $(a + b) \bmod n = (a \bmod n + b \bmod n) \bmod n$ и $(a \cdot b) \bmod n = (a \bmod n \cdot b \bmod n) \bmod n$, а онда ћемо их примењивати тако што ћемо производ (тј. збир) по модулу n свих претходних бројева применом функција комбиновати са новим бројем. Дакле, ако функцију за сабирање по модулу n означимо са $+_n$, а за множење по модулу n са $\cdot_n$, израчунаваћемо $((a +_n b) +_n c) +_n d$ тј. $((a \cdot_n b) \cdot_n c) \cdot_n d$.

Нагласимо да је израчунавање целобројног количника и остатка временски захтевна операција (најчешће се извршава доста спорије него основне аритметичке операције), тако да је у пракси пожељно избећи је када је то могуће. На пример, ако сте потпуно сигурни да ће $a + b$ бити могуће репрезентовати одабраним типом података, тада је уместо $(a \bmod n + b \bmod n) \bmod n$ ипак боље користити $(a + b) \bmod n$.

```
int plus_mod(int a, int b, int n) {
    return ((a % n) + (b % n)) % n;
}

int puta_mod(int a, int b, int n) {
    return ((a % n) * (b % n)) % n;
}

int main() {
    int a, b, c, d;
    cin >> a >> b >> c >> d;

    cout << plus_mod(plus_mod(plus_mod(a, b, 1000), c, 1000), d, 1000)
        << endl;
    cout << puta_mod(puta_mod(puta_mod(a, b, 1000), c, 1000), d, 1000)
        << endl;

    return 0;
}
```

Уместо у једном изразу, збир и производ можемо рачунати итеративно тако што их иницијализујемо на нулу тј. јединицу, а затим у сваком кораку текући збир мењамо збиром по модулу текућег збира и текућег броја, а текући производ мењамо производом по модулу текућег производа и текућег броја.

```

#include <iostream>
#include <cmath>

using namespace std;

int zbir_po_modulu(int a, int b, int n) {
    return ((a % n) + (b % n)) % n;
}

int proizvod_po_modulu(int a, int b, int n) {
    return ((a % n) * (b % n)) % n;
}

int main() {
    int a, b, c, d;
    cin >> a >> b >> c >> d;

    int zbir;
    zbir = zbir_po_modulu(a, b, 1000);
    zbir = zbir_po_modulu(zbir, c, 1000);
    zbir = zbir_po_modulu(zbir, d, 1000);
    cout << zbir << endl;

    int proizvod;
    proizvod = proizvod_po_modulu(a, b, 1000);
    proizvod = proizvod_po_modulu(proizvod, c, 1000);
    proizvod = proizvod_po_modulu(proizvod, d, 1000);
    cout << proizvod << endl;

    return 0;
}

```

Задатак: Монопол

Монопол је игра у којој се играчи крећу по пољима која су постављена у круг. Играчи се увек крећу у смеру казаљке на сату. Претпоставимо да су поља означена редним бројевима који крећу од 0, да су на почетку игре оба играча на том пољу и да се током игре играчи нису кретали унатраг. Ако се зна број поља које је први играч прешао од почетка игре и број поља које је други играч прешао од почетка игре напиши програм који израчунава колико корака први играч треба да направи да би дошао на поље на ком се налази други играч.

Улаз: Са стандардног улаза се уносе три природна броја. У првој линији дат је број поља на табли, у другој број поља које је од почетка игре прешао први, а у трећој број поља које је од почетка игре прешао други играч.

Излаз: На стандардни излаз треба исписати колико корака унапред играч треба да направи да би стигао на жељено поље.

Пример 1		Пример 2	
Улаз	Излаз	Улаз	Излаз
10	4	10	6
3		7	
7		3	

Решење

Означимо укупан број поља са n , број поља које је први играч прешао са A и број поља које је други играч прешао са B . Означимо број поља на коме се први играч тренутно налази са a , а број поља на које треба да стигне са b . Током свог кретања он је прешао k_a пуних кругова ($k_a \geq 0$) у којима је прешао по n поља и након тога још a поља, тако да је $A = k_a \cdot n + a$. Дакле, пошто је $0 \leq a < n$, на основу дефиниције целобројног дељења важи да је $a = A \bmod n$. Слично је и $b = B \bmod n$.

Ако је $b \geq a$ тада се број корака може израчунати као $b - a$. Међутим, могуће је и да важи $b < a$ и у том

случају играч мора прећи преко поља Start које је означено бројем 0. Број корака које играч треба да направи да би од поља на ком се налази стигао до поља Start је $n - a$, а број корака потребних да од поља Start стигне до жељеног поља је b , тако да је укупан број корака једнак $n - a + b$. На основу ове дискусије једноставно је направити програм који анализом ова два случаја стиже до решења.

```
int brojPolja;
int presaoPrvi, presaoDrugi;
cin >> brojPolja >> presaoPrvi >> presaoDrugi;
int saPolja = presaoPrvi % brojPolja;
int naPolje = presaoDrugi % brojPolja;
cout << (naPolje >= saPolja ?
        naPolje - saPolja :
        naPolje - saPolja + brojPolja)
    << endl;
```

Још једно гледиште које доводи до истог решења је могућност да се у случају да је $b < a$ промене ознаке на пољима иза старта. Тако би се поље Start уместо са 0 означило са n , следеће би се уместо са 1 означило са $1 + n$ итд., док би се поље b означило са $b + n$. Дакле, решење добијамо тако што умањилац увећавамо за n уколико је мањи од умањеника, и израчунавамо разлику.

```
int brojPolja;
int presaoPrvi, presaoDrugi;
cin >> brojPolja >> presaoPrvi >> presaoDrugi;
int saPolja = presaoPrvi % brojPolja;
int naPolje = presaoDrugi % brojPolja;
if (naPolje < saPolja)
    naPolje += brojPolja;
cout << naPolje - saPolja << endl;
```

Посматрано још из једног угла у овом задатку се тражи да се одреди разлика бројева B и A по модулу n . То сугерише да је уместо анализе случајева решење могуће добити израчунавањем вредности израза $(b - a + n) \bmod n$ тј. $(B \bmod n - A \bmod n + n) \bmod n$. Заиста, ако је $b \geq a$, вредност израза $b - a + n$ биће већа или једнака n и њен остатак при дељењу са n биће једнак вредности $b - a$ (тражење остатка ће практично поништити првобитно додавање вредности n). Са друге стране, ако је $b < a$ тада ће $b - a$ бити негативан број, па ће се додавањем вредности n добити број из интервала између 0 и $n - 1$. Зато проналажење остатка неће на крају променити резултат и добиће се вредност $b - a + n$ за коју смо рекли да је тражена вредност у овом случају.

```
int brojPolja;
int presaoPrvi, presaoDrugi;
cin >> brojPolja >> presaoPrvi >> presaoDrugi;
int saPolja = presaoPrvi % brojPolja;
int naPolje = presaoDrugi % brojPolja;
cout << (naPolje - saPolja + brojPolja) % brojPolja << endl;
```

Задатак: Разбрајалица

Деца стоје у кругу и одређују ко ће од њих да жмури тако што изговарају неку разбрајалицу (на пример, еци-печи-печ). Ако децу обележимо бројевима од 0 до $n - 1$, редом како стоје у кругу, ако је познат број речи (слогова) разбрајалице и ако је познат редни број детета од којег почиње разбрајање, напиши програм који приказује редни број детета које је одабрано да жмури (тј. на коме се разбрајање завршава).

Улаз: Уносе се три природна броја. Број слогова разбрајалице, број деце и редни број детета од којег почиње разбрајање. Претпоставићемо да је број деце мањи од 100, а да је број слогова мањи од 1000.

Издаз: Исписује се један природни број – редни број детета одабраног да жмури.

Пример

Улаз	Издаз
13	1
7	
3	

Објашњење

Претпоставимо да седморо деце у кругу користи разбрајалицу еци-пеци-пец, а да бројање креће од детета број 3, разбрајање ће тећи овако:

еци-пеци-пец-јасам-мали-зец-тиси-мала-препе-лица-еци-пеци-пец
 3 4 5 6 0 1 2 3 4 5 6 0 1

Решење

Нека је број слогова разбрајалице s , број деце d , а број детета од којег почиње бројање p . Ако би деца била поређана у једну дугачку врсту уместо у круг, тада би се бројање завршило на детету са бројем $p + (s - 1)$. Заиста, први слог се изговара на детету број p , други на детету број $p + 1$, трећи на детету број $p + 2$, а s -ти на детету $p + (s - 1)$.

Ако сада замислимо да се деца из те дугачке врсте ређају у круг у коме има тачно 7 места, сваком детету у врсти ће број његовог места у кругу бити додељен на следећи начин.

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

Приметимо да место у кругу које ће бити додељено детету које је у врсти на позицији k одређује тако што се одреди остатак тог броја k са 7 (на пример, остатак при дељењу броја 19 са 7 је 5 јер је $2 \cdot 7 + 5 = 19$, а броју 19 додељено је тачно место 5 на кругу). Слично ће важити и ако је број деце d , једино што ће тада уместо остатка при дељењу са 7 бити потребно одредити остатак при дељењу са d . Пошто смо рекли да би изабрано дете имало редни број $p + (s - 1)$ у врсти, у кругу ће то бити дете са редним бројем $(p + (s - 1)) \bmod d$.

О прекорачењу

Пошто важи да је $(a + b) \bmod c = (a \bmod c + b \bmod c) \bmod c$, што је доказано у задатку **Операције по модулу**, резултат би могао бити израчунат и изразом $(p \bmod d + (s - 1) \bmod d) \bmod d$, тј. $(p + (s - 1) \bmod d) \bmod d$, јер је $p < d$, па је $p \bmod d = p$. Овим би се донекле ублажила могућност настајања прекорачења у сабирању, ако би вредности p и s биле јако велике (што у овом задатку није случај).

```
int broj_slogova, broj_dece, prvo_dete;
cin >> broj_slogova >> broj_dece >> prvo_dete;
cout << (prvo_dete + (broj_slogova - 1)) % broj_dece << endl;
```

Задатак: Збир бројева по модулу

Напиши програм који израчунава збир природних бројева од 1 до n по датом модулу m .

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

- n ($10^2 \leq n \leq 2^{32} - 1$)
- m ($2 \leq m \leq 100$)

Израз: На стандардни излаз исписати један цео број који представља тражени збир по модулу.

Пример

Улаз	Израз
100	50
100	

Објашњење

Збир свих бројева од 1 до 100 је 5050 и остатак при дељењу са 100 је 50.

Решење

Наиван начин да се овај задатак реши је грубом силом, помоћу петље у којој се обрађују један по један број i од 1 до n и у чијем се телу се израчунава збир по модулу m дотадашњег збира z и текућег броја i . Како смо видели у задатку **Операције по модулу** то се може израчунати помоћу $(z \bmod m + i \bmod m) \bmod m$, тј. са $(z + i \bmod m) \bmod m$, јер у у сваком кораку важи $0 \leq z < m$, па је $z \bmod m = z$. Узевши у обзир ограничење вредности n дато у тексту задатка, потребно је употребити 64-битни тип података за представљање података.

```
typedef unsigned long long ull;
ull n, modul;
```

```
cin >> n >> modul;
ull zbir_po_modulu = 0;
for (ull i = 1; i <= n; i++)
    zbir_po_modulu = (zbir_po_modulu + i % modul) % modul;
cout << zbir_po_modulu << endl;
```

Ефикасно решење је да се примени формула за израчунавање збира аритметичког низа тј. Гаусова формула за збир првих n природних бројева (види задатак **Спортске припреме**) на основу које је $z = \frac{n(n+1)}{2}$. У овом задатку се тражи да се збир израчуна по модулу датог броја m . Тачно један од бројева n и $n + 1$ је паран, тако да се тражени резултат може израчунати као производ бројева $\frac{n}{2}$ и $n + 1$ по модулу m или као производ бројева n и $\frac{n+1}{2}$ по модулу m . Производ бројева a и b по модулу израчунавамо на основу формуле $((a \bmod m) \cdot (b \bmod m)) \bmod m$ коју смо приказали у задатку **Операције по модулу**.

```
typedef unsigned long long ull;

ull proizvod_po_modulu(ull a, ull b, ull m) {
    return ((a % m) * (b % m)) % m;
}

int main() {
    ull n, modul;
    cin >> n >> modul;
    ull zbir_po_modulu =
        n % 2 == 0 ?
        proizvod_po_modulu(n / 2, n + 1, modul) :
        proizvod_po_modulu(n, (n + 1) / 2, modul);
    cout << zbir_po_modulu << endl;
    return 0;
}
```

Задатак: Сат

Претпоставимо да приликом исказивања времена за сате користимо само бројеве од 1 до 12. Тако за 17:45 кажемо 5:45, а за 0:32 кажемо да је 12:32. Ако је познат број сати и минута такав да је број сати у опсегу од 0 до 23, а минута од 0 до 59, исказати то време тако да је број сати у опсегу 1 до 12.

Улаз: Учитавају се два броја, сваки у посебној линији. У првој линији је број сати (између 0 и 23), а у другој линији је број минута (између 0 и 59).

Издаз: Исписује се једна линија у којој се приказује временски тренутак у формату h:m, где је број сати h између 1 и 12, а број минута између 0 и 59.

Пример 1		Пример 2	
Улаз	Издаз	Улаз	Издаз
0	12:32	5	5:35
32		35	

Решење

У задатку је кључно одредити пресликавање сати, док се минути исписују непромењено. Пошто бројање треба да буде од 1 до 12, потребно је да реализујемо следеће пресликавање:

0	1	2	...	11	12	13	14	15	...	22	23	h
12	1	2	...	11	12	1	2	3	...	10	11	$f(h)$

Један од начина да се то уради је да се употреби гранање и да се вредност 0 преслика у вредност 12, да се вредности између 1 и 12 пресликају саме у себе, а да се од вредности веће од 12 пресликају одузимањем броја 12 од њих.

```
int h, m; // broj sati i minuta
cin >> h >> m;
// izracunavanje i ispis rezultata
```



```

if (h == 0)
    h = 12;
else if (h > 12)
    h = h - 12;
cout << h << ":" << m << endl;

```

Сати се периодично понављају на сваких 12 сати. Ако бисмо сате бројали од 0 до 11, тада би се одговарајуће пресликавање могло одредити једноставним проналажењем остатка при дељењу са 12.

0	1	2	...	11	12	13	14	15	...	22	23	h
0	1	2	...	11	0	1	2	3	...	10	11	$h \bmod 12$

Приметимо да након израчунавања остатка при целобројном дељењу са 12 проблем праве једино вредности 0 и 12 које су се пресликале у 0 уместо у 12.

Тај случај је могуће посебно испитати и поправити резултат.

```

int h, m; // broj sati i minuta
cin >> h >> m;
// izracunavanje i ispis rezultata
h = h % 12;
if (h == 0) h = 12;
cout << h << ":" << m << endl;

```

Ипак, циљ нам је да покажемо како је могуће решити овај задатак и без употребе гранања. Да бисмо добили вредност из интервала од 1 до 12, можемо на добијени остатак који је из интервала 0 до 11 додати 1. Међутим, да би пресликавање било коректно, потребно је ово увећање неутрализовати тако што ћемо пре одређивања остатка од полазног броја одузети 1. Дакле, број сати можемо одредити као $(h - 1) \bmod 12 + 1$. На пример, ако од 12 одузмемо 1, одредимо остатак добијеног броја 11 при дељењу са 12 и на добијени број 11 додамо 1 добићемо исправну вредност 12. Слично, ако од 13 одузмемо 1, одредимо остатак добијеног броја 12 са 12 и на добијени број 0 додамо 1 добићемо исправан резултат 1. Дакле, решење се реализује кроз следећа пресликавања

0	1	2	...	11	12	13	14	15	...	22	23	h
-1	0	1	...	10	11	12	13	14	...	21	22	$h - 1$
11	0	1	...	10	11	0	1	2	...	9	10	$(h - 1) \bmod 12$
12	1	2	...	11	12	1	2	3	...	10	11	$(h - 1) \bmod 12 + 1$

Једини број који може да прави проблем је 0 јер се почетним одузимањем броја 1 добија негативни број -1, а не желимо да се ослањамо на понашање остатка у случају негативног делиоца. То можемо поправити тако што на полазни број увек на почетку додамо 12 чиме обезбеђујемо да ће број увек бити позитиван, при чему се остатак при дељењу са 12 не мења. Дакле, тражено пресликавање је $(h + 12 - 1) \bmod 12 + 1$ или $(h + 11) \bmod 12 + 1$. Нагласимо да се ова техника додавања вредности модула користи када год је потребно одузети две вредности, по неком модулу. Наиме, подсетимо се да важи

$$(a - b) \bmod n = (a \bmod n - b \bmod n + n) \bmod n.$$

```

int h, m; // broj sati i minuta
cin >> h >> m;
// izracunavanje i ispis rezultata
cout << (h + 11) % 12 + 1 << ":" << m << endl;

```

Задатак: Навијање сата

Дочепо се мали брата очевога сата. Пошто је још мали, зна да гледа само сатну казаљку. Сат се може навијати тако што се једним дугметом сатна казаљка помера за један сат унапред (у смеру кретања казаљке на сату) а другим дугметом се помера за један сат уназад (у супротном смеру кретања казаљке на сату). Ако се зна позиција a на којој се тренутно налази казаљка, позиција b на којој брата жели да се нађе, напиши програм који одређује да ли се сат може навити тако што брата притисне једно од два дугмета тачно k пута.

2.1. МОДУЛАРНА АРИТМЕТИКА

Улаз: Са стандардног улаза учитавају се бројеви $0 \leq a, b \leq 11$ и број $0 \leq k \leq 1000$.

Изназ: На стандардни излаз написати текст `napred` ако се сат може навити тако што се казаљка помера унапред, `nazad` ако се сат може навити тако што се казаљка помера уназад (ако је сат могуће навити на оба начина, исписати `napred`, а затим `nazad` у следећем реду) тј. `ne moze` ако се сат не може навити.

Пример 1

Улаз	Изназ
1	napred
7	nazad
18	

Пример 2

Улаз	Изназ
2	nazad
4	
10	

Пример 3

Улаз	Изназ
7	ne moze
9	
3	

Решење

Потребно посебно проверити да ли се сат може навити померањем казаљке напред, и померањем казаљке назад. Померање казаљке за 12 корака у било којем смеру враћа казаљку на почетни положај. Стога је померање за k корака могуће ако и само ако је могуће померање за $k \bmod 12$ корака и на почетку је број k могуће заменити бројем $k \bmod 12$.

Померање казаљке унапред за k корака доводи је са положаја a на положај $(a + k) \bmod 12$, док је померање уназад за k корака доводи на положај $(a - k + 12) \bmod 12$ (сабирање и одузимање по модулу описали смо у задатку [Операције по модулу](#) и [Монопол](#)). Ако је први број једнак b , тада се казаљка може померити унапред, а ако је други број једнак b , тада се казаљка може померити уназад. Ако ниједно од та два није испуњено, тада се казаљка може померити (за проверу овог услова можемо употребити логичку променљиву коју иницијализујемо на вредност `false` и постављамо је на `true` ако је испуњен први, а затим и други услов).

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
    int a, b, k;
    cin >> a >> b >> k;
    k = k % 12;
    bool moze = false;
    if ((a + k) % 12 == b) {
        moze = true;
        cout << "napred" << endl;
    }
    if ((a - k + 12) % 12 == b) {
        moze = true;
        cout << "nazad" << endl;
    }
    if (!moze)
        cout << "ne moze" << endl;
    return 0;
}
```

Решење којим се може одредити положај казаљке након k померања унапред, а које не користи модуларну аритметику се заснива на томе да положај a у петљи увећавамо за 1 тачно k пута, при чему је враћамо на нулу када достигне вредност 12. Слично, померање уназад би се реализовало умањивањем вредности a за 1 у петљи тачно k пута, при чему је враћамо на 11 када постане негативна. Наравно, овакво решење може бити веома неефикасно ако се на почетку k не замени са $k \bmod 12$.

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
    int a, b, k;
    cin >> a >> b >> k;

    bool moze = false;
```

```

int a1 = a;
for (int i = 0; i < k; i++) {
    a1++;
    if (a1 == 12)
        a1 = 0;
}
if (a1 == b) {
    moze = true;
    cout << "napred" << endl;
}

int a2 = a;
for (int i = 0; i < k; i++) {
    a2--;
    if (a2 < 0)
        a2 = 11;
}
if (a2 == b) {
    moze = true;
    cout << "nazad" << endl;
}

if (!moze)
    cout << "ne moze" << endl;

return 0;
}

```

Задатак: Степен по модулу

Напиши програм који израчунава степен по модулу, тј. за дате бројеве a , n и m израчунава $a^n \bmod m$.

Улаз: Свака од три линије стандардног улаза садржи бројеве a , n и m , при чему је $1 \leq a, n, m \leq 4 \cdot 10^9$.

Издаз: На стандардни издаз испиши тражену вредност степена по модулу.

Пример

Улаз	Издаз
2	376
100	
1000	

Решење

Директно решење подразумева степеновање помоћу узастопног множења бројем a (производ иницијализујемо на 1 и n пута га множимо са a). С обзиром на распон из ког долазе бројеви, јасно је да ће степен бити изузетно велики број и да неће моћи да се представи основним типовима података. Стога је много боље током самог множења израчунавати остатак по модулу тј. множење вршити по формули $(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$. Пошто су бројеви $a \bmod m$ и $b \bmod m$ 32-битни, њихов производ може бити 64-битан, тако да је пре израчунавања остатка, пре множења чиниоце конвертовати у 64-битни тип података. Ако након сваког множења вршимо израчунавање остатка при дељењу са m , није неопходно пре множења проналазити остатак при дељењу (први чинилац биће текући степен који ће већ бити у интервалу $[0, m)$, док је други чинилац a , који је истог реда величине као и m , па нам проналажења остатка не значи пуно).

Иако коректан, овај алгоритам је преспор, с обзиром на велики број корака множења који се врше, с обзиром на величину изложноца n . Наиме, сложеност овог алгоритма је $O(n)$.

```
#include <iostream>
```

```
using namespace std;
```

2.1. МОДУЛАРНА АРИТМЕТИКА

```
int main() {
    unsigned a, n, mod;
    cin >> a >> n >> mod;

    a %= mod;

    unsigned s = 1;
    for (unsigned i = 0; i < n; i++)
        s = ((unsigned long long)s * (unsigned long long)a) % mod;

    cout << s << endl;

    return 0;
}
```

Ефикасније решење можемо постићи алгоритмом брзог степеновања. Најједноставнија је рекурзивна формулација.

- Ако је $n = 0$, резултат је 1.
- Ако је n паран број, тада је $a^n = a^{2 \cdot n/2} = (a^2)^{n/2}$. Слично, важи да је $a^n \bmod m = a^{2 \cdot n/2} \bmod m = (a^2 \bmod m)^{n/2} \bmod m$.
- Ако је n непаран број, тада је $a^n = a \cdot a^{n-1}$. Слично важи да је $a^n \bmod m = (a \cdot (a^{n-1} \bmod m)) \bmod m$.

Нагласимо да се пре израчунавања a^2 , као и пре израчунавања $a \cdot (a^{n-1} \bmod m)$, број a мора конвертовати у 64-битни тип података.

Пошто се после највише два рекурзивна позива вредност изложивоца смањи барем два пута сложеност овог алгоритма је $O(\log n)$.

```
#include <iostream>

using namespace std;

// izracunavamo (a^n) % mod
unsigned stepen(unsigned a, unsigned n, unsigned mod) {
    if (n == 0)
        return 1;
    else if (n % 2 == 0) {
        unsigned long long aull = a;
        return stepen((aull * aull) % mod, n/2, mod);
    } else {
        unsigned long long aull = a;
        return (aull * stepen(a, n-1, mod)) % mod;
    }
}

int main() {
    unsigned a, n, mod;
    cin >> a >> n >> mod;
    cout << stepen(a % mod, n, mod) << endl;
    return 0;
}
```

У претходом решењу могуће је ослободити се рекурзије, посматрајући бинарни запис изложивоца n .

```
#include <iostream>

using namespace std;

// izracunavamo (a^n) % mod
unsigned stepen(unsigned a, unsigned n, unsigned mod) {
```

```

unsigned s = 1;
while (n > 0) {
    if ((n & 1) == 0) {
        a = ((unsigned long long)a * (unsigned long long)a) % mod;
        n = n >> 1;
    } else {
        s = ((unsigned long long)s * (unsigned long long)a) % mod;
        n--;
    }
}
return s;
}

int main() {
    unsigned a, n, mod;
    cin >> a >> n >> mod;
    cout << stepen(a % mod, n, mod) << endl;
    return 0;
}

```

2.2 Теорија бројева

Задатак: Савршени бројеви

Број је савршен ако је једнак суми својих делилаца (у збир делилаца броја се убраја број 1, али не и сам тај број). На пример, број 28 је савршен јер су му делиоци 1, 2, 4, 7 и 14, важи да је $1+2+4+7+14=28$. Написати програм који одређује најмањи савршен број у интервалу $[n - k, n + k]$.

Улаз: Са стандардног улаза уноси се природан број n ($1 \leq n \leq 1000000$) и k ($1 \leq k < n$).

Излаз: На стандардни излаз исписати тражени број, ако постоји тј. текст NE ако у интервалу нема савршених бројева.

Пример

Улаз	Излаз
400	496
100	

Решење

Линеарна претрага свих делилаца

Из саме формулације задатка, јасно је да је његова важна компонента функција која за произвољни број n рачуна збир његових делилаца. Најдиректније решење да се збир делилаца броја x израчуна је да се редом, у петљи, за сваки број провери да ли је дилац броја n и ако јесте да се увећа текући збир делилаца.

Остаје питање који су бројеви потенцијални делиоци тј. које бројеве треба проверити у петљи. Пошто је формулацијом задатка одређено да се рачуна број 1, а да се не рачуна сам број n , најгрубље је да се провере сви природни бројеви већи и једнаки 1, а строго мањи од n . Међутим, можемо урадити и мало боље од тога. Последњи потенцијални дилац броја који је мањи од њега је број $\frac{n}{2}$. Заиста, ако би постојао дилац i који је строго већи од $\frac{n}{2}$ и строго мањи од n , онда би $\frac{n}{i}$ такође био дилац, који је строго мањи од 2, а строго већи од 1, што није могуће. Зато је у петљи довољно проверавати целе бројеве између 1 и $\frac{n}{2}$ тј. $\lfloor \frac{n}{2} \rfloor$ што се може израчунати као $n/2$.

Сложеност овог приступа провере за један број n је $O(n)$.

У главном програму примењујемо алгоритам претраге. Логички индикатор који указује да ли је пронађен савршен број иницијализујемо на вредност `false`, а затим у петљи пролазимо кроз све бројеве од $n - k$ до $n + k$ и проверавамо да ли су савршени. Први пут када наиђемо на савршен број постављамо индикатор на `true`, памтимо текућу вредност бројачке променљиве и одмах прекидамо петљу (на пример, наредбом `break`), јер се тражи најмањи број у интервалу. На крају, на основу вредности идентификатора и упамћене вредности савршеног броја пријављујемо тражени резултат.

Укупна сложеност је, дакле, $O(k(n+k))$, а када је k ограничено са n , може се оценити и са $O(kn)$ тј. $O(n^2)$.

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

// zbir delilaca broja n (racunajuci 1, ali ne racunajuci n)
int zbirDelilaca(int n) {
    int zbir = 1; // zbir inicijalizujemo na prvi delioc (to je broj 1)
    // proveravamo potencijalne delioce izmedju 2 i n/2.
    // Jedini delilac veci od n/2 je n, a njega ne treba racunati.
    for (int i = 2; i <= n/2; i++)
        if (n % i == 0) // broj i jeste delilac, pa ga treba dodati na sumu
            zbir += i;
    return zbir;
}

// proverava da li je broj savrsen
bool savrsen(int n) {
    return zbirDelilaca(n) == n;
}

int main() {
    // učitavamo ulazne podatke
    int n, k;
    cin >> n >> k;

    // algoritmom linearne pretrage proveravamo da li u intervalu
    // [n-k, n+k] postoji savrsen broj
    bool postojiSavrsen = false;
    int savrsenBroj;
    for (int i = n-k; i <= n+k; i++) {
        if (savrsen(i)) {
            // savrsen broj je pronadjen
            savrsenBroj = i;
            postojiSavrsen = true;
            break;
        }
    }
    // prikazujemo rezultat
    if (postojiSavrsen)
        cout << savrsenBroj << endl;
    else
        cout << "NE" << endl;
    return 0;
}
```

Линеарна претрага свих парова делилаца

Ни овако дефинисан програм не достиже жељену брзину и може се додатно убрзати. Користимо чињеницу да се делиоци броја увек јављају у пару. Ако је i делилац броја n , онда знамо да је и број $\frac{n}{i}$ такође делилац броја n . Делиоцу 1 одговара број n , евентуалном делиоцу 2 одговара евентуалан делиоц $\frac{n}{2}$ итд. На пример, делиоци броја 100 су 1, 2, 4, 5, 10, 20, 25, 50 и 100 и они се јављају у паровима (1, 100), (2, 50), (4, 25) (5, 20) и (10, 10). Ако је први делилац мањи од квадратног корена броја n , онда је други делилац већи од квадратног корена броја n . За сваки делилац који је већи од корена броја n постоји делилац који је мањи од корена броја n и зато је анализу делилаца довољно вршити до вредности корена из n . Ако је број потпун квадрат, онда је и корен делилац и он сам себи одговара као пар. Пошто не желимо корен два пута да додамо на збир, случај испитивања корена броја ћемо издвојити и извршити након петље која ће се вршити све док је број кандидат за делиоца строго мањи од корена. Такође, пошто сам број не желимо да рачунамо у збир, а број један желимо, суму ћемо иницијализовати на 1, а петља ће кренути од вредности 2.

Сложеност овог приступа за проверу једног броја n је $O(\sqrt{n})$, а провера свих савршених бројева је у интервалу $[n - k, n + k]$, када је $k < n$ се може ограничити са $O(k\sqrt{n})$ тј. $O(n\sqrt{n})$.

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

// zbir delilaca broja n (racunajuci 1, ali ne racunajuci n)
int zbirDelilaca(int n) {
    int zbir = 1; // jedan je delilac svakog broja
    int koren = (int)sqrt(n); // ceo deo korena iz n
    for (int i = 2; i <= koren; i++)
        if (n % i == 0) {
            // ako je i delilac broja n, delilac je i n div i
            zbir += i;
            zbir += n / i;
        }
    // slucaj korena posebno obrađujemo (ako je delilac, u petlji je
    // dodat dva puta umesto jednom, pa ga treba oduzeti)
    if (koren*koren == n)
        zbir -= koren;
    return zbir;
}

bool savrsen(int n) {
    return zbirDelilaca(n) == n;
}

int main() {
    // učitavamo ulazne podatke
    int n, k;
    cin >> n >> k;

    // algoritmom linearne pretrage proveravamo da li u intervalu
    // [n-k, n+k] postoji savrsen broj
    bool postojiSavrsen = false;
    int savrsenBroj;
    for (int i = n-k; i <= n+k; i++) {
        if (savrsen(i)) {
            // savrsen broj je pronadjen
            savrsenBroj = i;
            postojiSavrsen = true;
            break;
        }
    }
    // prikazujemo rezultat
    if (postojiSavrsen)
        cout << savrsenBroj << endl;
    else
        cout << "NE" << endl;
    return 0;
}
```

Растављање на просте чиниоце

Још боље решење је да се збир делилаца израчуна на основу растављања броја на просте чиниоце које приказујемо у задатку **Растављање на просте чиниоце**. Број облика p^n , где је p прост фактор, има делиоце $1, p, p^2, \dots, p^n$ и њихов збир је $1 + p + \dots + p^n$. Важи да је збир делилаца броја мултипликативна функција.

тј да је збир делилаца производа два узајамно проста броја једнак производу збира делилаца сваког од њих (покушајте ово да докажете). Ако се број може раставити на просте чиниоце у облику $p_1^{n_1} \dots p_k^{n_k}$, тада је његов збир делилаца је једнак $(1 + p_1 + \dots + p_1^{n_1}) \cdot \dots \cdot (1 + p_k + \dots + p_k^{n_k})$, јер су степени свих простих фактора међусобно узајамно прости. Ако знамо вредност израза $1 + p_i + \dots + p_i^{n_i-1}$, тада се вредност израза $1 + p_i + \dots + p_i^{n_i}$ може добити додавањем сабирка $p_i^{n_i}$, али и много брже, множењем полазне вредности бројем p_i и додавањем броја један.

Током рада алгоритма одржавамо текући збир делилаца који иницијализујемо на 1 (зато што ће бити рачунат у облику производа збинова делилаца појединачних простих фактора). У петљи проверавамо све потенцијалне просте чиниоце од броја 2, па све док текући кандидат не пређе вредност корена текућег броја n . У сваком кораку одржавамо збир делиоца текућег простог фактора који такође иницијализујемо на 1 јер ћемо га рачунати као производ. Док год је број n дељив бројем p (што проверавамо у посебној, унутрашњој петљи) збир делилаца текућег простог фактора множимо са p и додајемо му 1 док број n делимо са p . На крају, тела спољашње петље укупан збир делилаца множимо збиром делилаца управо обрађеног простог фактора (то може бити и 1, ако број n није био дељив бројем p) и увећавамо број p . По изласку из петље је могуће да је број n остао већи од 1, што значи да он садржи последњи прост фактор и тада се укупан збир делилаца множи вредношћу $1 + n$ (што је збир делилаца простог броја n).

Овако одређени збир и број делилаца укључују и сам број n који је накнадно потребно одузети да би се добио збир описан у услови задатка.

Када на располагању имамо функцију која рачуна збир делилаца броја, тада је функцију која проверава да ли је број савршен тривијално дефинисати (она само за дати број позива функцију која рачуна збир делилаца и проверава да ли је он једнак броју).

Сложеност израчунавања збира делилаца одговара сложености факторизације која је једнака $O(\sqrt{f_0})$, где је f_0 највећи прост фактор броја n . Оно је најспорије за просте бројеве, за које је $f_0 = n$ и сложеност обраде сваког простог броја је $O(\sqrt{n})$. Међу бројевима $[n - k, n + k]$ највећи прости фактори се разликују, али свакако се сложеност може ограничити са $O(k\sqrt{n})$ (мада ће се за многе бројеве радити и мање операција него што ова груба процена указује).

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

// zbir delilaca broja n (racunajuci 1, ali ne racunajuci n)
int zbirDelilaca(int n) {
    int n0 = n;    // pamtimo polaznu vrednost broja n
    int zbir = 1;   // zbir delilaca kao proizvod
    int p = 2;      // prvi kandidat za delioca
    while (p*p <= n) {
        // zbir delilaca broja faktora p^i u prostoј faktorizaciji
        int zbir_pi = 1;
        while (n % p == 0) {           // ekstrahujemo faktor p^i
            zbir_pi = zbir_pi * p + 1; // azuriramo tekuci proizvod
            n /= p;                     // delimo broj njime
        }
        zbir *= zbir_pi;                // azuriramo globalni zbir
        p++;                           // prelazimo na sledeceg kandidata
    }
    if (n > 1)                          // preostao je jedan prost faktor
        zbir *= (1 + n);
    return zbir - n0;                  // polazna vrednost broja n nije potrebna
}

// proverava da li je broj savrsen
bool savrsen(int n) {
    return zbirDelilaca(n) == n;
}

int main() {
    // učitavamo ulazne podatke
```



```

int n, k;
cin >> n >> k;

// algoritmom linearne pretrage proveravamo da li u intervalu
// [n-k, n+k] postoji savrsen broj
bool postojiSavrsen = false;
int savrsenBroj;
for (int i = n-k; i <= n+k; i++) {
    if (savrsen(i)) {
        // savrsen broj je pronadjen
        savrsenBroj = i;
        postojiSavrsen = true;
        break;
    }
}
// prikazujemo rezultat
if (postojiSavrsen)
    cout << savrsenBroj << endl;
else
    cout << "NE" << endl;
return 0;
}

```

Изрчунавање свих савршених бројева унапред

На крају рецимо и да су савршени бројеви карактеристични по томе што их је веома мало. У првој милијарди то су само бројеви 6, 28, 496, 8128 и 33550336. На основу овог знања, задатак се могао решити и мало “кварно” тако што би се у функцији која испитује да ли је број савршен он просто упоредио са овим бројевима.

Пошто се провера да ли је број савршен врши у сложености $O(1)$, сложеност овог решења је $O(k)$.

```

#include <iostream>
using namespace std;

// provera da li je broj savrsen
bool savrsen(int n) {
    return n == 6 || n == 28 || n == 496 || n == 8128 || n == 33550336;
}

int main() {
    // učitavamo ulazne podatke
    int n, k;
    cin >> n >> k;

    // algoritmom linearne pretrage proveravamo da li u intervalu
    // [n-k, n+k] postoji savrsen broj
    bool postojiSavrsen = false;
    int savrsenBroj;
    for (int i = n-k; i <= n+k; i++) {
        if (savrsen(i)) {
            // savrsen broj je pronadjen
            savrsenBroj = i;
            postojiSavrsen = true;
            break;
        }
    }
    // prikazujemo rezultat
    if (postojiSavrsen)
        cout << savrsenBroj << endl;
}

```

```

else
    cout << "NE" << 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 ($1 \leq a \leq b \leq 500000$).

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

Пример 1		Пример 2	
Улаз	Израз	Улаз	Израз
1	6 6	300000	308620 389924
1000	28 28 220 284 496 496	400000	356408 399592

Решење

Провера свих парова бројева

Слично као у задатку **Савршени бројеви** и у овом задатку је кључна функција она која израчунава збир делилаца датог броја. Њену имплементацију можемо потпуно преузети из тог задатка.

Што се тиче проналажења пријатељских бројева, на први поглед може деловати да треба испитати сваки пар бројева $i \leq j$ из интервала $[a, b]$, и проверити да ли је збир делилаца од i једнак j и да ли је збир делилаца од j једнак i .

Сложеност најгорег случаја одређивања броја делилаца за број n наступа када је број n прост и једнака је $O(\sqrt{n})$.

С обзиром на потенцијално велику ширину интервала, испитивање свих парова бројева траје прилично дуго. Такав алгоритам испитује квадратни број парова односу на ширину интервала $b - a$, и у наивном случају квадратни број пута израчунава збир делилаца броја тј. сложеност јој је се грубо може проценити на $O((b - a)^2 \sqrt{b})$. Чак иако би се зборови делилаца свих бројева унапред израчунали, алгоритам би и даље вршио квадратни број поређења (сложеност израчунавања свих зборови била би $O((b - a) \sqrt{b})$, а провера $O((b - a)^2)$).

```
#include <iostream>
```

```
using namespace std;
```

```

// zbir delilaca broja n (racunajuci 1, ali ne racunajuci n)
int zbirDelilaca(int n) {
    int n0 = n; // pamtimo polaznu vrednost broja n
    int zbir = 1; // zbir delilaca kao proizvod
    int p = 2; // prvi kandidat za delioca
    while (p*p <= n) {
        // zbir delilaca broja faktora p^i u prostoј faktorizaciji
        int zbir_pi = 1;
        while (n % p == 0) { // ekstrahujemo faktor p^i
            zbir_pi = zbir_pi * p + 1; // azuriramo tekuci proizvod
            n /= p; // delimo broj njime
        }
        zbir *= zbir_pi; // azuriramo globalni zbir
        p++; // prelazimo na sledeceg kandidata
    } // broj racunara koji nas zanima
    if (n > 1) // preostao je jedan prost faktor

```

```

    zbir *= (1 + n);
    return zbir - n0;           // polazna vrednost broja n nije potrebna
}

// Brojevi su prijateljski ako je suma delilaca prvog jednaka drugom
// broju i obratno. Funkcija pronalazi i stampa sve razlicite
// neuređene parove prijateljskih brojeva u datom intervalu [a, b]
void prijateljskiBrojevi(int a, int b) {
    // proveravamo sve parove brojeva
    for (int i = a; i < b; i++)
        for (int j = i; j <= b; j++)
            if (zbirDelilaca(i) == j && zbirDelilaca(j) == i)
                cout << i << " " << j << endl;
}

int main() {
    int a, b;
    cin >> a >> b;
    prijateljskiBrojevi(a, b);
    return 0;
}

```

Провера јединственог потенцијалног пријатеља

Кључна опаска за оптимизацију је да сваки број има јединственог потенцијалног пријатеља (то је број који је једнак збиру његових делилаца). Дакле, нема потребе да анализирамо све парове тј. да проверавамо све могуће вредности j да бисмо међу њима пронашли вредност j која је једнака збиру делилаца броја i .

Суштински у програму филтрирамо серију бројева из интервала $[a, b]$ на основу услова да је збир делилаца збира делилаца броја једнака њему самом. Зато се за сваки број i из интервала $[a, b]$ израчунава потенцијални пријатељ j (једнак збиру делилаца броја i), а онда се проверава да ли j припада интервалу $[a, b]$, да ли је $i \leq j$ и на крају, да ли је и збир делилаца броја j једнак броју i .

Овај алгоритам врши линеаран број израчунавања функције за збир делилаца (у односу на ширину интервала $b - a$) и сложеност му се грубо може проценити са $O((b - a)\sqrt{b})$.

Генерално, испитивање да ли се у интервалу $[a, b]$ налази број x никада нема смисла радити помоћу петље и испитивања свих појединачних елемената тог интервала.

```

#include <iostream>

using namespace std;

// zbir delilaca broja n (racunajuci 1, ali ne racunajuci n)
int zbirDelilaca(int n) {
    int n0 = n;           // pamtimo polaznu vrednost broja n
    int zbir = 1;         // zbir delilaca kao proizvod
    int p = 2;            // prvi kandidat za delioca
    while (p*p <= n) {
        // zbir delilaca broja faktora p^i u prostoј faktorizaciji
        int zbir_pi = 1;
        while (n % p == 0) {           // ekstrahujemo faktor p^i
            zbir_pi = zbir_pi * p + 1; // azuriramo tekuci proizvod
            n /= p;                     // delimo broj njime
        }
        zbir *= zbir_pi;               // azuriramo globalni zbir
        p++;                           // prelazimo na sledeceg kandidata
    } // broj racunara koji nas zanima
    if (n > 1)                         // preostao je jedan prost faktor
        zbir *= (1 + n);
    return zbir - n0;                 // polazna vrednost broja n nije potrebna
}

```

```

}

// Brojevi su prijateljski ako je suma delilaca prvog jednaka drugom
// broju i obratno. Funkcija pronalazi i stampa sve razlicite
// neuređene parove prijateljskih brojeva u datom intervalu [a, b]
void prijateljskiBrojevi(int a, int b) {
    for (int i = a; i <= b; i++) { // proveravamo sve brojeve do maksimuma
        // jedini potencijalni prijatelj broja i je njegov zbir delilaca
        int j = zbirDelilaca(i);
        if (a <= j && j <= b && i <= j && zbirDelilaca(j) == i)
            cout << i << " " << j << endl;
    }
}

int main() {
    int a, b;
    cin >> a >> b;
    prijateljskiBrojevi(a, b);
    return 0;
}

```

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 ($1 \leq n \leq 10^9$).

Издаз: На стандардни издаз исписати DA ако је број n прост тј. NE ако није.

Пример		Пример 2	
Улаз	Издаз	Улаз	Издаз
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. 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` (тима се уједно прекида извршавање петље). На крају петље, функција може да врати `true`, јер није пронађен ниједан делилац 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 потенцијалних 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 задатку **Ератостеново сито**.

Задатак: Најближи прост број

Аутор задатака за такмичење треба да састави тест пример за програм који испитује да ли је дати број прост. Пошто жели да испита програм на примерима разне тежине, потребно је да одреди неки прост број који је близу милијарде, неки који је близу милиона и слично. Помози му тако што ћеш написати програм који одређује најближи прост број унетом броју.

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

Излаз:

- Ако је број n прост, испиши поруку `prost` број и вредност броја n одвојене размаком.
- Ако постоје два броја која су на истом растојању од броја n испиши поруку `dva` броја и просте бројеве, одвојене размаcima. Исписати прво мањи, па онда већи прост број.
- Ако је јединствен најближи прост број мањи од броја n испиши поруку `manji` број и вредност тог броја, раздвојене размаком.
- Ако је јединствен најближи прост број већи од броја n испиши поруку `veci` број и вредност тог броја, раздвојене размаком.

Пример 1

Улаз Излаз
12 dva broja 11 13

Пример 2

Улаз Излаз
19 prost broj 19

Пример 3

Улаз Излаз
24 manji broj 23

Решење

Функцију којом се проверава да ли је дати број прост можемо преузети из задатка **Прост број**. Са том функцијом на располагању, имплементација решења представља малу вежбу алгоритма линеарне претраге и логичких услова и гранања. Након читавања броја n проверавамо да ли је он прост. Ако јесте приказујемо одговарајућу поруку, а ако није, онда започињемо претрагу за најближим простим бројевима тако што ћемо у првом кораку проверавати бројеве $n - 1$ и $n + 1$, у другом $n - 2$ и $n + 2$, у трећем $n - 3$ и $n + 3$ и тако даље, све док не наиђемо на неки број који јесте прост. Дакле, одржавамо променљиву k која креће од 1 и у сваком кораку петље се увећава за 1, проверавајући при том бројеве $n - k$ и $n + k$. У сваком кораку петље непоходно је да и мањи и већи прост број буду проверени, да би се могла пријавити одговарајућа порука ако су оба та броја проста. Приликом умањивања треба бити обазриви и повести рачуна да број који се проверава не буде негативан (пре провере броја проверавамо да ли је $n \geq k$). Петља се завршава када се деси да је неки од бројева $n - k$ или $n + k$ прост.

```
#include <iostream>
```

```
using namespace std;
```

```
// funkcija koja proverava da li je dati broj prost
```

```
bool prost(int n) {
    if (n <= 1) return false;    // brojevi <= 1 nisu prosti
    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;

    if (prost(n))
        cout << "prost broj " << n << endl;
    else {
        bool manjiJeProst = false; // jos nije pronaden prost broj manji od n
        bool veciJeProst = false; // jos nije pronaden prost broj veci od n
        int k = 1; // rastojanje od broja n
        // dok nismo nasli prost broj manji ili veci od datog
        while (!manjiJeProst && !veciJeProst) {
            if (n >= k && prost(n - k)) // da li je n-k pozitivan i prost broj
                manjiJeProst = true;
            if (prost(n + k)) // da li je n+k prost broj
                veciJeProst = true;

            if (manjiJeProst && veciJeProst)
                // dva prosta broja su na istom rastojanju
                cout << "dva broja" << " " << n - k << " " << n + k << endl;
            else if (manjiJeProst)
                // manji broj je prost
                cout << "manji broj " << n - k << endl;
            else if (veciJeProst)
                // veci broj je prost
                cout << "veci broj " << n + k << endl;
            k++; // uvecavamo rastojanje
        }
    }
    return 0;
}
```

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	168 76127
1000	

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 $(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$. То је детаљно описано и коришћено у задатку **Операције по модулу**. 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 ($zbir = (zbir \% 1000000 + p \% 1000000) \% 1000000$). 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() {
    // učitavamo 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

2.2. ТЕОРИЈА БРОЈЕВА

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

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

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 користимо претходну оцену.

Још је велики Ојлер открио да се збир $H(m) = 1 + 1/2 + 1/3 + \dots + 1/m = \sum_{d \leq m} \frac{1}{d}$ (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 $\log \log \sqrt{n} = \log \log n^{\frac{1}{2}} = \log (\frac{1}{2} \log n) = \log \frac{1}{2} + \log \log n$, 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 Ератостеново 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() {
    // učitavamo 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 угнеђеним 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 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: Велики прости бројеви

Напиши програм који исписује све просте бројеве међу унетима. Програм може да погреши у малом броју случајева, али треба да ради брзо.

Улаз: Свака линија стандардног улаза (има их највише 1000) садржи један природан број мањи од 10^{15} .

Издаз: На стандардни издаз исписати оне унете бројеве који су прости.

Пример

Улаз	Издаз
7916413003241	7916413003241
16819606497999	33711348088423
33711348088423	60894005190391
99222770171192	
60894005190391	
4917349288929	

Решење

Провера делилаца

За сваки учитани број можемо проверити да ли је прост, коришћењем провере делилаца, тј. поступака описаних у задатку **Прост број**.

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

typedef uint64_t ull;
typedef unsigned __int128 uLL;

ull stepen(ull a, ull n, ull m) {
    if (n == 0)
        return 1;
    if (n % 2 == 0)
        return (stepen(((uLL)(a % m) * (uLL)(a % m)) % m, n/2, m)) % m;
    return ((a % m)*stepen(a, n-1, m)) % m;
}

bool fermaovTest(unsigned long long p, int k) {
    if (p == 1) return false;
    if (p == 2 || p == 3) return true;
    default_random_engine gen;
    uniform_int_distribution<unsigned> dist(2, p-2);
    for (int i = 0; i < k; i++) {
        unsigned long long a = dist(gen);
        if (stepen(a, p-1, p) != 1)
            return false;
    }
    return true;
}

int main() {
    unsigned long long a;
    while (cin >> a)
        if (fermaovTest(a, 100))
            cout << a << endl;
}
```

Фермаов тест прималности

Један веома једноставан пробабилистички тест прималности заснован је на малој Фермаовој теореме која тврди да за прост број p и произвољан број a који није дељив бројем p важи

$$a^{p-1} \bmod p = 1.$$

На пример, ако је $p = 7$ и $a = 4$, важи да је $4^6 = 4096 = 585 \cdot 7 + 1$ тј. да је $4^6 \bmod 7 = 1$.

Један веома леп, а крајње елементаран начин да се докаже мала Фермаова теорема је да се размотре све ниске дужине p у којима се јављају слова из азбуке која има a симбола. Таквих ниски има a^p . На пример, ако је $p = 3$ и $a = 2$, то су ниске XXX , XXY , XYX , XYX , YXX , YXY , YYX и YYY . Замислимо сада да су слова сваке ниске написана на кругу. Тиме све дате ниске делимо у одређене групе тако да две ниске припадају истој групи ако и само ако се добијају читањем слова са истог круга. У једној групи је само XXX , у другој су XXY , XYX и YXX , у трећој групи су XYX , YXY и YXY , док је у последњој, четвртој групи само YYY . У општем случају постојаће a група које су једночлане и у њима ће бити ниске које садрже p понављања једног истог симбола из азбуке. Све остале групе имаће по p елемената. Наиме, ако би се запис дат на неком кругу могао прочитати на два иста начина кренувши од два различита слова, тада би тај запис морао бити периодичан и период би морао да дели дужину ниске. Међутим, пошто је p прост број, његови једини делиоци су 1 и p , тако да свака група има или 1 или p различитих елемената. Дакле, важи да је $a^p = x \cdot p + a$, где је x број група са по p елемената, да је $a^p - a = x \cdot p$ тј. да је $a(a^{p-1} - 1) = x \cdot p$. Пошто a није дељиво са p и не може да има заједничких простих фактора са њим (јер је p прост), следи да $a^{p-1} - 1$ мора да буде дељиво са p , што је урпаво тврђење мале Фермаове теореме.

Напреднији докази засновани су на Лагранжовој теорему теорије група. Наиме, остаци по модулу p тј. бројеви $\{1, 2, \dots, p-1\}$ чине групу при операцији множења по модулу p (налажење инверза сваког елемента описано је у задатку [Модуларни инверз](#)). Ред било ког елемента a из овог скупа је најмањи број k такав да је $a^k \bmod p = 1$ и на основу Лагранжове теореме ред елемента мора да дели ред групе тј. број њених елемената који је $p-1$. Дакле, важи да је $a^{p-1} \bmod p = a^{km} \bmod p = (a^k)^m \bmod p = (a^k \bmod p)^m \bmod p = 1^m \bmod p = 1$.

Мала Фермаова теорема може се схватити и као специјални случај Ојлерове теореме (коју описујемо у задатку [Модуларни инверз](#) и [Ојлерова функција](#)) која каже да је $a^{\phi(n)} \bmod n = 1$, где је $\phi(n)$ број елемената који су узајамно прости са бројем n . Ако је n прост број и ако га обележимо са p , тада је број узајамно прости бројева у односу на њега једнак $p-1$, одакле директно следи мала Фермаова теорема.

Пошто изложилац може бити велики, степен по модулу морамо рачунати алгоритмом брзог степеновања, како је описано у задатку [Степен по модулу](#).

Да бисмо израчунали $a^2 \bmod p$, можемо применити формулу $((a \bmod p) \cdot (a \bmod p)) \bmod p$, међутим, и у том случају морамо множити два 64-битна броја (јер због ограничења да је $p < 10^{15}$, за његову репрезентацију морамо употребити 64-битне бројеве). Производ два 64-битна броја може лако прекорачити опсег 64-битних бројева и за репрезентацију овог производа, пре проналажења остатка при дељењу са p , морамо употребити бар 128-битни тип података.

У стандардном језику C++ немамо такав тип података, међутим, преводиоци GCC и Clang нам на 64-битних архитектурама рачунара пружају типове `__int128` и `unsigned __int128`.

```
#include <iostream>
```

```
using namespace std;
```

```
bool prost(unsigned long long n) {
    if (n == 1 ||
        (n % 2 == 0 && n != 2) ||
        (n % 3 == 0 && n != 3))
        return false;
    for (unsigned long long 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() {
    unsigned long long a;
    while (cin >> a)
        if (prost(a))
            cout << a << endl;
}
```



```
    return 0;
}
```

Задатак: Број простих у интервалима

Напиши програм који брзо може да утврди колико у датим интервалима природних бројева има простих.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10000$) који представља број интервала, затим, у наредних n линија по два броја a и b ($1 \leq a < b \leq 10^6$) који представљају крајеве затвореног интервала $[a, b]$.

Излаз: На стандардни излаз исписати n природних бројева (сваки у посебној линији) који представљају број простих бројева у сваком интервалу $[a, b]$.

Пример

Улаз	Излаз
3	25
1 100	143
100 1000	1061
1000 10000	

Решење

Директно решење

Наиван начин да се задатак реши је да се учитава интервал по интервал $[a, b]$, да се у петљи анализирају сви бројеви од a до b , да се за сваки помоћном функцијом проверава да ли је прост (такву функцију смо видели, на пример, у задатку **Прост број**), и да се броје они који то задовољавају. Велики проблем овог решења је то што ће се због очекиваног преклапања интервала за пуно бројева више пута испитивати да ли су прости. Сложеност овог поступка у најгорем случају је $O(n \cdot m \cdot \sqrt{m})$ где је n број интервала, а m највећи број у интервалима. С обзиром на ограничења $n \leq 10^4$, а $m \leq 10^6$, јасно је да би овај поступак одузео пуно времена.

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

using namespace std;

// funkcija ispituje da li je broj prost
bool prost(int n) {
    if (n < 2) return false;
    for (int i = 2; i*i <= n; i++)
        if (n % i == 0)
            return false;
    return true;
}

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

    // broj intervala
    int n;
    cin >> n;

    for (int i = 0; i < n; i++) {
        // učitavamo interval
        int a, b;
        cin >> a >> b;
        // brojimo proste brojeve u njemu
        int brojProstih = 0;
        for (int i = a; i <= b; i++)
            if (prost(i))
```

```
    brojProstih++;  
    // ispisujemo rezultat  
    cout << brojProstih << '\n';  
}  
  
return 0;  
}
```

Ератостеново сито

Ератостеновим ситом које смо приказали у задатку **Ератостеново сито** могуће је много брже имплементирати истовремену проверу да ли су сви бројеви до неке дате границе прости. У почетној фази се креира низ логичких променљивих тако да је на позицији i у том низу вредност `true` ако и само ако је број i прост. Сито можемо примењивати за сваки интервал појединачно, али је много боље ако одредимо максимални десни крај свих интервала и направимо сито до тог броја (након тога ћемо за сваки број у свим интервалима моћи у константном времену да проверимо да ли је прост). У овом сценарију потребно је да прво складиштимо све интервале, да бисмо се могли вратити на њихову обраду када им одредимо максимални десни крај.

Поступак заснован на Ератостеновом ситиу је много бољи него понављање провере за сваки број појединачно. Сложеност сита се може проценити као $O(m \cdot \log \log m)$, па се укупна сложеност процењује као $O(m \cdot \log \log m + n \cdot m)$ - сито се изврши једном, а затим се n пута проверава по највише m бројева, при чему је свака та провера константне сложености. Фактор $\log \log m$ је веома мали, тако да се сложеност може проценити и са $O(n \cdot m)$, што је опет превише споро за ограничења дата у овом задатку. Напоменимо да смо меморијску сложеност програма подигли за простор потребан да се складишти $O(m)$ логичких променљивих.

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
  
using namespace std;  
  
void Eratosten(int n, vector<bool>& prosti) {  
    // alociramo potreban prostor  
    prosti.resize(n + 1, true);  
    prosti[0] = prosti[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 (prosti[i]) {  
            // precrtavamo umnoske broja i i to krenuvsi od i*i  
            for (int j = i * i; j <= n; j += i)  
                prosti[j] = false;  
        }  
}  
  
int main() {  
    ios_base::sync_with_stdio(false);  
  
    // broj intervala  
    int n;  
    cin >> n;  
  
    // učitavamo sve intervale  
    typedef pair<int, int> interval;  
    vector<interval> intervali(n);  
    for (int i = 0; i < n; i++)  
        cin >> intervali[i].first >> intervali[i].second;
```

```

// odredjujemo maksimalni desni kraj intervala
int max = max_element(intervali.begin(), intervali.end(),
    [](interval p1, interval p2) {
        return p1.second < p2.second;
    })->second;

// za sve brojeve iz [0, max] odredjujemo koji su brojevi prosti
vector<bool> prosti;
Eratosten(max, prosti);

// obradjujemo sve intervale
for (auto interval : intervali) {
    // izracunavamo broj prostih brojeva u tekucem intervalu
    int brojProstih = 0;
    for (int i = interval.first; i <= interval.second; i++)
        if (prosti[i])
            brojProstih++;
    // ispisujemo rezultat
    cout << brojProstih << endl;
}

return 0;
}

```

Разлагање интервала на разлику два интервала-префикса

Дакле, критично је убрзати корак израчунавања броја простих бројева у интервалу $[a, b]$, чак и када се провера да ли је сваки појединачни број прост врши веома брзо. Кључна идеја је она која се може видети и у задацима **Аритметички троугао** или **Сегмент датог збира у низу целих бројева**. Просте бројеве у интервалу $[0, b]$ можемо разложити на просте бројеве који припадају интервалу $[0, a - 1]$ и просте бројеве који припадају интервалу $[a, b]$. Пошто су ова два скупа дисјунктна, број простих бројева у интервалу $[a, b]$ једнак је разлици између броја простих бројева у интервалу $[0, b]$ и броја простих бројева у интервалу $[0, a - 1]$. Дакле, ако за сваку вредност x до највећег десног краја израчунамо број простих бројева у интервалу $[0, x]$, за сваки интервал $[a, b]$ број простих бројева можемо добити у константном времену. Мала мана овога је што ангажујемо додатни простор од m целих бројева (мада је то могуће редукovati, ако је потребно, тако што памтимо само вредности придружене крајевима наших интервала). Број простих бројева у интервалима $[0, x]$ опет се може израчунавати инкрементално (слично као што смо, на пример, збирове префикса рачунали инкрементално у задатку **Префикс највећег збира**). Ови бројеви чине рекурентну серију и број бројева у интервалу $[0, x]$ је исти као број бројева у интервалу $[0, x - 1]$ ако x није прост број, тј. за један је већи од броја простих бројева у интервалу $[0, x - 1]$, ако x јесте прост. Ако низ добијен Ератостеновим ситом замислимо као низ индикатора (нула и јединица), онда се број простих бројева у сваком интервалу $[0, x]$ заправо израчунава као низ парцијалних сума низа индикатора. Да бисмо израчунали парцијалне суме потребно нам је $O(m)$ операција, што укупну сложеност своди на $O(m \cdot \log \log m + m + n)$, што је практично линеарна сложеност.

```

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

using namespace std;

// Funkcija koja Eratostenovim sitom za svaki broj iz intervala
// [0, n] odredjuje da li je prost
void Eratosten(int n, vector<bool>& prosti) {
    // alociramo potreban prostor
    prosti.resize(n + 1, true);
    prosti[0] = prosti[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 (prosti[i]) {  
        // precrtavamo umnoske broja i i to krenuvsi od i*i  
        for (int j = i * i; j <= n; j += i)  
            prosti[j] = false;  
    }  
}  
  
int main() {  
    ios_base::sync_with_stdio(false);  
  
    // broj intervala  
    int n;  
    cin >> n;  
  
    // učitavamo sve intervale  
    typedef pair<int, int> interval;  
    vector<interval> intervali(n);  
    for (int i = 0; i < n; i++)  
        cin >> intervali[i].first >> intervali[i].second;  
  
    // odredjujemo maksimalni desni kraj ucitanih intervala  
    int max = 0;  
    for (auto interval : intervali)  
        if (interval.second > max)  
            max = interval.second;  
  
    // za sve brojeve iz [0, max] odredjujemo koji su brojevi prosti  
    vector<bool> prosti;  
    Eratosten(max, prosti);  
  
    // brojProstihDo[i] = broj prostih brojeve iz intervala [0, i]  
    vector<int> brojProstihDo(max + 1);  
    brojProstihDo[0] = 0;  
    for (int i = 1; i <= max; i++)  
        brojProstihDo[i] = brojProstihDo[i-1] + prosti[i];  
  
    // za svaki ucitani interval  
    for (auto interval : intervali) {  
        // broj prostih iz [a, b] je razlika broja prostih iz [0, b] i  
        // broja prostih iz [0, a-1]  
        int brojProstih = brojProstihDo[interval.second] -  
            brojProstihDo[interval.first - 1];  
        cout << brojProstih << endl;  
    }  
  
    return 0;  
}
```

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 ($1 \leq n \leq 2 \cdot 10^9$).

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

Пример

Улаз	Изназ
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 пријављујући 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 прост 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 променљиве n прост број (а видећемо да за то није потребно чекати да вредност f достигне n).

Прикажимо рад алгоритма на једном примеру.

n	f	$cinloc$
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 променљиве 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 променљиве 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>

using namespace std;

int main() {
    // učitavamo broj koji treba rastaviti na proste ciniocce
    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;
}
```

Избегавање угнежђених петљи

Можда је мало изненађујуће, али у претходном решењу се могу избећи угнежђене петље тако што се вредност f у петљи увећава само ако n није био дељив са f (ако јесте, можда ће бити дељив још једном, тако да f задржава своју вредност да би се у наредном кораку петље опет проверила дељивост њоме).

```
#include <iostream>

using namespace std;

int main() {
    // učitavamo broj koji treba rastaviti na proste ciniocce
    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) {
        if (n % f == 0) { // pronađen je cinilac f koji mora biti prost
            cout << f << " "; // prijavljujemo pronađeni prost cinilac
            n /= f; // delimo broj njime
        }
        else // f nije prost cinilac broja n
            f = f + 1;
    }
    cout << endl;
    return 0;
}
```

```

    f = f + 1;           // prelazimo na sledeceg kandidata
}
cout << endl;
return 0;
}

```

Поправљање ефикасности

Кључни проблем претходног алгоритма је то што се зауставља тек када се број сведе на 1. У тренутку када је вредност променљиве n постане проста, вредност делиоца f се увећава све док не достигне вредност n . Међутим, на основу задатка **Прост број** знамо да је број прост ако и само ако нема делилаца већих од свог корена. Дакле, ако се покаже да је f веће од квадратног корена броја n , то значи да је текућа вредност броја n прост број и не треба даље тражити његове просте чиниоце (вредност n садржи последњи прост чинилац полазног броја n_0).

Прикажимо рад алгоритма на једном примеру.

n	f	cinioс
3300	2	2
1650	2	2
825	2	-
825	3	3
275	3	-
275	4	-
275	5	5
55	5	5
11	5	-
		11

Претрага престаје када је $f^2 > n$ тј. када је $5^2 > 11$. Тада знамо да је 11 прост број и он је последњи чиниоц.

Пошто укупан број чинилаца можемо сматрати практично константним, сложеност овог алгоритма је $O(\sqrt{M})$, где је M највећи прост чинилац полазног броја.

```
#include <iostream>
```

```
using namespace std;
```

```

int main() {
    // učitavamo broj koji treba rastaviti na proste cinioce
    int n;
    cin >> n;
    int f = 2; // prvi potencijalni prost cinilac je 2
    // dok se deljenjem sa prostim činiocima broj n ne svede na neki prost ili 1
    while (f*f <= n) {
        while (n % f == 0) { // dok je f prost cinilac broja n
            cout << f << " "; // prijavljujemo pronađeni prost cinilac
            n /= f;           // delimo broj njime
        }
        f = f + 1;           // prelazimo na sledeceg kandidata
    }
    if (n > 1)              // prijavljujemo n - preostali prost cinilac
        cout << n << " ";
    cout << endl;
    return 0;
}

```

Задатак: Допуна до пуног квадрата

Напиши програм који за унети природни број n одређује најмањи број m такав да је $n \cdot m$ потпун квадрат.

Улаз: Са стандардног улаза се уноси природни број n ($1 \leq n \leq 2 \cdot 10^{12}$).

Излаз: На стандардни излаз исписати тражени број m .

Пример

Улаз	Излаз
104	26

Решење

Најмањи број којим се n множи да би се добио потпун квадрат мора бити делилац броја n (он мора да буде производ простих фактора броја n који се јављају са непарном вишеструкошћу). Зато можемо испитати све делиоце броја n (линеарном претрагом) и проверити који је први који када се помножи са n даје потпун квадрат. Проверу да ли је број потпун квадрат можемо вршити уз помоћ функције (реалног) кореновања (додуше, то може некада довести до грешака услед непрецизности у запису реалних бројева).

Пошто се делиоци јављају увек у пару (ако је број d делилац броја n , онда је делилац и број n/d), претрагу можемо вршити само до вредности $\sqrt{n}$. Анализа делилаца у пару је приказана у задатку **Савршени бројеви**.

Ако је број $d \leq \sqrt{n}$ први који задовољава услов, он је најмањи такав и претрага се може прекинути. Са друге стране, када број n/d задовољи услов, само то региструјемо и настављамо претрагу, јер не знамо да је он најмањи такав.

Осим што може бити донекле неефикасна, основни проблем са оваквом имплементацијом је то што број n и његови делиоци могу бити 64-битни бројеви, па се њихов производ не може представити исправно 64-битним бројем, тако да овај приступ може да да гарантовано исправне резултате само ако је $n \leq 10^9$.

Сложеност овог приступа је $O(\sqrt{n})$.

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

bool jeKvadrat(long long k) {
    long long koren = (long long)sqrt(k);
    return koren * koren == k;
}

int main() {
    long long n;
    cin >> n;
    long long najmanji = 1;
    for (long long d = 1; d * d <= n; d++) {
        if (n % d == 0) {
            if (jeKvadrat(n * d)) {
                najmanji = d;
                break;
            }
            if (jeKvadrat(n * (n / d)))
                najmanji = n / d;
        }
    }
    cout << najmanji << endl;
    return 0;
}
```

Размотримо број 234. Он се може раставити на просте чиниоце као $2 \cdot 3 \cdot 3 \cdot 13$. Да би овај број био потпун квадрат потребно је да се сваки чинилац понови паран број пута и зато је тражена допуна $2 \cdot 13$. Заиста, множењем са $2 \cdot 13 = 26$ добија се број $2^2 \cdot 3^2 \cdot 13^2$ који је квадрат броја $2 \cdot 3 \cdot 13 = 78$.

Када се пун квадрат растави на просте чиниоце, сваки чинилац се јавља паран број пута. Ако је број $n = p_1^{k_1} \cdot p_j^{k_j}$, тада је тражена допуна једнака производу оних чинилаца p_i за који је вишеструкоост k_i непаран број. Дакле, задатак се једноставно решава применом алгоритма растављања на просте чиниоце. Тај алгоритам је описан у задатку **Растављање на просте чиниоце**. Издвајамо чинилац по чинилац броја и за сваки од њих бројимо вишеструкоост (тако што сваки пут када поделимо број n са неким простим чиниоцем p увећамо вред-

ност бројача k). Када утврдимо да неки чинилац има непарну вишеструкост, тада њиме množимо тренутну вредност броја m (коју иницијализујемо на 1).

Приметимо да нема потребе да израчунавамо производ бројева n и m , па је исправно све време користити 64-битни тип података (који је довољан за запис бројева n и m , али не и њиховог производа).

Сложеност овог алгорита одговара сложености растављања на просте чиниоце, а то је $O(p_{max})$, где p_{max} највећи прост чинилац броја n . Када је n велики прост број, сложеност је $O(\sqrt{n})$.

```
#include <iostream>

using namespace std;

long long dopunaDoPunogKvadrata(long long n) {
    // dopuna
    long long m = 1;

    // rastavljamo broj n na proste ciniocce

    // kandidat za prost cinilac
    long long p = 2;
    while (p * p <= n) {
        // racunamo visestrukost cinioca p
        int k = 0;
        while (n % p == 0) {
            n /= p;
            k++;
        }
        // ako se cinilac p javlja neparan broj puta tada se on mora
        // javiti u dopuni m
        if (k % 2 != 0)
            m *= p;

        // prelazimo na sledeceg kandidata
        p++;
    }

    // n se sveo na svoj najveći prost cinilac
    if (n > 1)
        // i on mora da ucestvuje u dopuni
        m *= n;

    // vracamo rezultat
    return m;
}

int main() {
    long long n;
    cin >> n;
    cout << dopunaDoPunogKvadrata(n) << endl;
    return 0;
}
```

Задатак: Производи потпуни квадрати

На табли је записан број 1. Имамо низ a од n природних бројева и у i -том кораку ($1 \leq i \leq n$) бришемо тренутни број на табли и уместо њега пишемо производ њега и броја a_i . После сваког корака одредити да ли је тренутни број на табли потпун квадрат.

Улаз: У првом реду стандардног улаза налази се природан број n који представља дужину низа a . У наредном реду налази n ($1 \leq n \leq 5 \cdot 10^5$) природних бројева (између 1 и милион) раздвојених размаком - то су елементи

низа a у редоследу којим множе тренутни број на табли.

Излаз: За сваки елемент низа a , у редоследу датим на улазу, исписати **da** уколико је његов производ са тренутним бројем на табли потпун квадрат а иначе исписати **ne**.

Пример

Улаз	Излаз
7	ne
2 3 6 15 35 21 64	ne
	da
	ne
	ne
	da
	da

Решење

С обзиром на то да не можемо да представимо производ коришћењем целобројног типа (чак ни помоћу 64-бита), уместо самог броја, чуваћемо списак његових простих чинилаца. Да би број био потпун квадрат, сваки прост чинилац треба да се јавља паран број пута. Можемо сматрати да се свака два идентична проста чиниоца “скраћују” тј. не утичу на то да ли је број потпун квадрат или није. Зато уместо свих простих чинилаца производа можемо чувати само оне који су преостали након таквог скраћивања (то су недостајући чиниоци и сваки од њих треба да се јави непаран број пута у наредним множењима да би број постао потпун квадрат). Те чиниоце можемо памтити у облику скупа (у језику C++ можемо користити `set` или `unordered_set`). (у језику C# можемо користити `SortedSet` или `HashSet`).

Када број раставимо на просте чиниоце (алгоритмом описаним у задатку [Растављање на просте чиниоце](#), анализирамо један по један чинилац и оне који тренутно нису у скупу додајемо у скуп, а оне који јесу, избацујемо из скупа. Број је пун квадрат ако и само ако је скуп празан након обраде свих чинилаца.

Илуструјмо на крају рад овог алгоритма на примеру.

i	ai	cinioci	skup	kvadrat
0	2	2	{2}	ne
	3	3	{3}	ne
	6	2, 3	{}	da
	15	3, 5	{3, 5}	ne
	35	5, 7	{3, 7}	ne
	21	3, 7	{}	da
	64	2, 2, 2, 2, 2, 2	{}	da

Сваки од n бројева растављамо на просте чиниоце, за шта нам је укупно потребно време $O(n\sqrt{n})$. Укупан број чинилаца је одређен горњим ограничењем елемената низа K (пошто је оно једнако 10^6 , сваки број највише може да има двадесетак простих чинилаца). Укупно је, дакле, око $O(n \log k)$ чинилаца. Пошто очекујемо да ће се скуп често смањивати услед уклањања елемената, сложеност убацивања и брисања из таквог скупа се може сматрати константном. Ограничавајући фактор је, дакле, факторизација (тј. $O(n\sqrt{n})$).

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

```
using namespace std;
```

```
vector<int> prostiCinioci(int n) {
    vector<int> cinioci;
    int d = 2;
    while (d * d <= n) {
        while (n % d == 0) {
            cinioci.push_back(d);
            n /= d;
        }
        d++;
    }
}
```

```

    if (n > 1)
        cinioci.push_back(n);
    return cinioci;
}

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);
    int n;
    cin >> n;
    set<int> nedostajuciCinioci;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        vector<int> cinioci = prostiCinioci(x);
        for (int c : cinioci) {
            auto it = nedostajuciCinioci.find(c);
            if (it != nedostajuciCinioci.end())
                nedostajuciCinioci.erase(it);
            else
                nedostajuciCinioci.insert(c);
        }
        cout << (nedostajuciCinioci.empty() ? "da" : "ne") << '\n';
    }
    return 0;
}

```

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 мравца, 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.

Наивна претрага

Један наиван начин да се одреди НЗД је да се употреби линеарна претрага (описана у задатку **Негативан број**) и да се редом провере сви бројеви од мањег од бројева a и b уназад до 1 и као резултат био би пријављен први који је делилац оба (пошто се бројеви ређају уназад, то би био највећи заједнички делилац).

Најгори случај наступа када су бројеви узајамно прости, када се вредност делиоца спушта до 1. Сложеност тог приступа је, дакле, $O(\min(a, b))$.

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

using namespace std;

// izracunavanje najveceg zajednickog delioca brojeva a i b
int nzd(int a, int b) {
    for (int d = min(a, b); d >= 1; d--)
        if (a % d == 0 && b % d == 0)
            return d;
    return 1; // necemo nikada stici dovde
}

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

Разлагање на просте чиниоце

Једна могућност која се класично спроводи када се ручно одређује НЗД, је да се примени факторизација оба броја и да се уоче заједнички прости чиниоци. Факторизација се врши основу поступка описаног у задатку **Растављање на просте чиниоце**.

Пошто се факторизација броја n може извршити у сложености $O(\sqrt{n})$, сложеност овог приступа је $O(\sqrt{a} + \sqrt{b})$ тј. $O(\sqrt{\max(a, b)})$. Пошто чинилаца може бити мало (с обзиром на дато ограничење бројева, највише тридесетак), можемо сматрати да се проналажење заједничких чинилаца извршава практично у константној сложености.

```
#include <iostream>

using namespace std;

int nzd(int a, int b) {
    // rezultat
    int n = 1;
    // tekuci faktor
    int d = 2;
    while (d*d <= a && d*d <= b) {
        // zajednicke proste faktore dodajemo na rezultat
        while (a % d == 0 && b % d == 0) {
            n *= d;
            a /= d; b /= d;
        }
        // uklanjamo preostala pojavljivanja faktora d koja nisu zajednicka
        while (a % d == 0)
            a /= d;
        while (b % d == 0)
            b /= d;
        // prelazimo na naredni faktor
        d++;
    }
    // manji od dva broja je prost - proveravamo da li je on poslednji
    // zajednicki prost faktor
    if (a % b == 0)
        n *= b;
}
```

```

else if (b % a == 0)
    n *= a;
return n;
}

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

```

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.

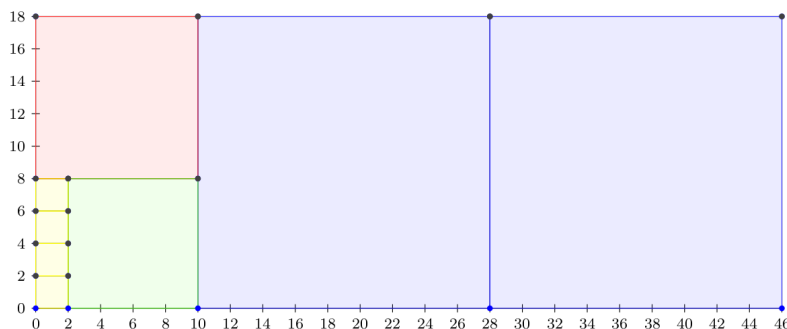
У супротном се проблем може свести на проналажење НЗД мањег од два броја и разлике два броја. На пример, ако је $a > b$, тада је $\text{nzd}(a, b) = \text{nzd}(a - b, b)$.

Заиста, ако неки број d дели бројеве a и b , тада он дели и њихову разлику. Дакле, $d = \text{nzd}(a, b)$ 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 $\text{nzd}(a, b) = \text{nzd}(a, b - a)$.

Алгоритам се може илустровати и геометријски (и у директној је вези са проблемом одређивања максималне димензије квадрата којима може да се попљоча 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 попљочати 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 попљочамо тај преостали правоугаоник, да ћемо тим квадратима успети да попљочамо и ове квадрате димензије 18 пута 18 (јер ће димензија тих малих квадрата делити број 18), па ћемо самим тим моћи попљочати 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 попљочати тај преостали правоугаоник ће бити такви да се њима може попљочати 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 попљочати 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.

$$\begin{aligned} nzd(279, 45) &= nzd(234, 45) = nzd(189, 45) = nzd(144, 45) = nzd(99, 45) = nzd(54, 45) = nzd(9, 45) = \\ nzd(9, 36) &= nzd(9, 27) = nzd(9, 18) = nzd(9, 9) = 9. \end{aligned}$$

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 $a = (a \div b) \cdot b + (a \bmod b)$. 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 Еуклидовога 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 променљиве 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 резонувањем, 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 резонувањем, 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 променљиве a и b треба да имају вредности чланова r_1 и r_2 , што значи да се пар променљивих 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 променљивама a и b , док b не постане нула и тада је НЗД који је једнак r_k садржан у променљивој 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() {
    // učitavamo 3 broja
    int a, b, c;
    cin >> a >> b >> c;
    // izracunavamo njihov nzd
    cout << nzd(nzd(a, b), c) << endl;
    return 0;
}
```

Задатак: Број дељив са 1 до n

Најмањи број који је дељив свим бројевима од 1 до 10 је број 2520. Напиши програм који за дати природан број n одређује који је најмањи број дељив бројевима од 1 до n .

Улаз: Са стандардног улаза учитава се број n ($1 \leq n < 23$).

Излаз: На стандардни излаз се исписује тражени број.

Пример

Улаз	Излаз
10	2520

Решење

Најмањи број који је дељив са датим бројевима је по дефиницији њихов најмањи заједнички садржалац. Израчунавање најмањег заједничког садржаоца два броја може се свести на израчунавање њиховог највећег заједничког делиоца ако се зна да је $nzs(a, b) = \frac{a \cdot b}{nzd(a, b)}$. Дакле, ако имплементирамо функцију `nzd` која рачуна највећи заједнички делиоц два броја Еуклидовим алгоритмом (као у задатку **Највећи заједнички делилац**), тада функцију `nzs` можемо имплементирати као

```
int nzs(int a, int b) {
    return (a / nzd(a, b)) * b;
}
```

Рецимо да је због опасности од настајања прекорачења прилично битно да се прво изврши дељење, па тек онда множење.

НЗС више од два броја рачунамо итеративно, тако што променљиву у којој га чувамо поставимо на 1, а затим у петљи обрађујемо број по број и променљиву мењамо вредношћу најмањег заједничког садржаоца њене текуће вредности и текућег броја.

```
#include <iostream>
```

```
using namespace std;
```

```
// najveći zajednicki delilac dva broja
```

```
int nzd(int a, int b) {
    while (b > 0) {
        int tmp = a;
        a = b;
        b = tmp % b;
    }
    return a;
}
```



```

}

// najmanji zajednicki sadrzalac dva broja
int nzs(int a, int b) {
    return (a / nzd(a, b)) * b;
}

int main() {
    // ucitavamo broj n
    int n;
    cin >> n;

    // racunamo nzs(1, 2, 3, ..., n)
    int m = 1;
    for (int i = 1; i <= n; i++)
        m = nzs(m, i);

    // ispisujemo rezultat
    cout << m << endl;
    return 0;
}

```

Задатак: Билијар из ћошка

Билијарски сто је правоугаоног облика димензије $m \times n$ и има четири рупе у ћошковима. Посматрајмо цртеж стола, такав да му је ширина m , а висина n . Лоптица се удара из доњег левог угла (поља са координатама $(0, 0)$) дуж линије која је под углом од 45° у односу на ивице стола. Ако претпоставимо да лоптица не успорава своје кретање, да се од сваке се ивице одбија под углом од 45° , да је веома мала и да у рупу упада само ако су јој координате центра једнаке координате рупе, напиши програм који одређује у коју рупу ће после неког времена упасти, као и колико ће се пута пре тога одбити о ивице стола.

Улаз: Са стандардног улаза се уносе два цела броја m и n ($1 \leq m, n \leq 10^9$) који представљају димензије стола.

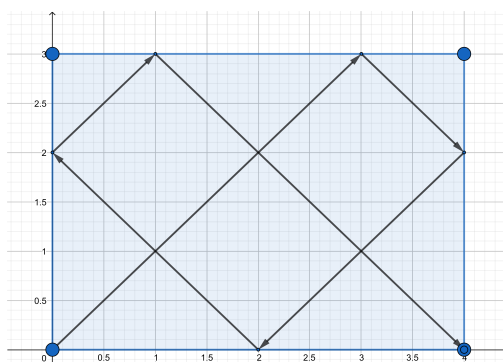
Излаз: На стандардни излаз у првом реду исписати координате рупе у коју ће лоптица упасти, а у другом број одбијања о ивице стола пре него што се то деси.

Пример

Улаз	Излаз
4 3	4 0

Објашњење

Кретање лоптице је приказано на слици.



Слика 2.1: Билијар

Решење

Да би се лоптица нашла у некој рупи она крећући се хоризонтално треба да пређе растојање (дужину пређеног пута) које представља неки умножак ширине стола и крећући се вертикално треба да пређе растојање које представља неки умножак висине стола. Пошто је интензитет њене брзине и у хоризонталном и у вертикалном правцу исти (јер се стално креће под углом од 45 степени у односу на неку ивицу стола), растојање пређено у хоризонталном и у вертикалном смеру је увек исто. Дакле, лоптица упада у рупу у тренутку када је пређено растојање први пут неки умножак бројева m и n . Најмањи број који је дељив и са m и са n је њихов најмањи заједнички садржалац $nzs(m, n)$.

Број пута n_x који је лоптица хоризонтално пребрисала целу ширину стола једнак је количнику $nzs(m, n)$ и броја m , а број n_y пута који је лоптица вертикално пребрисала целу висину стола једнак је количнику $nzs(m, n)$ и броја n . Ако је број n_x непаран, лоптица се налази у некој рупи на десној, а ако је паран, налази се у некој рупи на левој ивици стола, па x координату рупе лако одређујемо анализом парности броја n_x . Аналогно, анализом парности броја n_y одређујемо координату y рупе.

Ако је лоптица хоризонталну ширину прешла n_x пута, она се о неку леву и десну ивицу одбила тачно $n_x - 1$ пута. Слично, ако је вертикалну ширину прешла n_y пута, о горњу и доњу ивицу се одбила тачно $n_y - 1$ пута. Укупан број одбијања је, дакле, $(n_x - 1) + (n_y - 1)$.

НЗС два броја можемо израчунати Еуклидовим алгоритмом, али морамо водити рачуна да због великих улазних параметара може наступити прекорачење и морамо користити адекватне типове податка.

```
#include <iostream>
```

```
using namespace std;
```

```
int nzd(int a, int b) {
    while (b > 0) {
        int ost = a % b;
        a = b;
        b = ost;
    }
    return a;
}

long long nzs(int a, int b) {
    return ((long long)a / nzd(a, b)) * b;
}
```

```
int main() {
    int m, n;
    cin >> m >> n;
    long long S = nzs(m, n);
    int nx = S / m, ny = S / n;
    int x = nx % 2 == 0 ? 0 : m;
    int y = ny % 2 == 0 ? 0 : n;
    cout << x << " " << y << endl;
    cout << (nx - 1) + (ny - 1) << endl;
    return 0;
}
```

Рачунање НЗС можемо избећи. За решење нама је потребно да знамо вредности $n_x = nzs(m, n)/m$ и $n_y = nzs(m, n)/n$. Пошто је $nzs(m, n) = (mn)/nzd(m, n)$, важи да је $n_x = n/nzd(m, n)$ док је $n_y = m/nzd(m, n)$. Тиме избегавамо потребу за радом са великим бројевима и избегавамо могућност настанка прекорачења.

```
#include <iostream>
```

```
using namespace std;
```

```
int nzd(int a, int b) {
    while (b > 0) {
```

```

    int ost = a % b;
    a = b;
    b = ost;
}
return a;
}

int main() {
    int m, n;
    cin >> m >> n;
    int N = nzd(m, n);
    int nx = n / N, ny = m / N;
    int x = nx % 2 == 0 ? 0 : m;
    int y = ny % 2 == 0 ? 0 : n;
    cout << x << " " << y << endl;
    cout << (nx - 1) + (ny - 1) << endl;
    return 0;
}

```

Задатак: Ученици на истим седиштима

Ученици гледају два филма у биоскопској сали у којој су седишта распоређена у m врста и n колона (ученика има тачно $m \cdot n$). Када су гледали први филм, наставници су их распоредили тако што су их ређали по азбучном редоследу попуњавајући врсту по врсту, а када су гледали други филм, поново су били распоређени по азбучном редоследу, али овај пут врсту по врсту. Напиши програм који одређује колико ученика је седело на истом месту током гледања оба филма.

Улаз: Са стандардног улаза се уносе два цела броја m и n ($1 \leq m, n \leq 10000$), сваки у посебном реду.

Излаз: На стандардни излаз исписати један цео број који представља број ученика који су током гледања оба филма седели на истом седишту.

Пример

Улаз	Излаз
3	3
5	

Решење

Приликом ређања елемената врсту по врсту при чему у једној врсти има n елемената, елемент са редним бројем x налази у врсти $v_1 = x \div n$ и колони $k_1 = x \bmod n$ (бројање елемената, врста и колона почиње од нуле). Дакле на основу дефиниције целобројног дељења важи да је $x = v_1 \cdot n + k_1$, $0 \leq k_1 < n$. Слично, ако се елементи ређају колону по колону, и ако у једној колони има m елемената, тада се елемент са редним бројем x налази у колони $k_2 = x \div m$ и врсти $v_2 = x \bmod m$. Дакле, важи да је $x = k_2 \cdot m + v_2$ и $0 \leq v_2 < m$.

Да би елемент био на истој позицији мора да важи да је $k_1 = k_2$ и да је $v_1 = v_2$. Једно директно решење је да се за сваки редни број ученика од 0 до $m \cdot n - 1$ одреде v_1 , k_1 , v_2 и k_2 (целобројним дељењем, на горе описани начин) и затим да се види колико тако одређених бројева задовољава тражени услов (решење се, дакле, заснива на бројању елемената филтриране серије).

```

#include <iostream>

using namespace std;

int main() {
    int m, n;
    cin >> m >> n;
    int brojNaIstimSedistima = 0;
    for (int ucenik = 0; ucenik < m*n; ucenik++) {
        int v1 = ucenik / m, k1 = ucenik % m;
        int k2 = ucenik / n, v2 = ucenik % n;
        if (v1 == v2 && k1 == k2)

```

```
    brojNaIstimSedistima++;  
}  
cout << brojNaIstimSedistima << endl;  
return 0;  
}
```

Међутим, постоји и ефикасније решење. На основу горе реченог, мора да важи да је $x = v_1 \cdot n + k_1 = k_2 \cdot m + v_2$, а пошто x задовољава услов задатка ако и само ако важи $k_1 = k_2$ и $v_1 = v_2$, важи да је $v_1 \cdot n + k_1 = k_1 \cdot m + v_1$, тј. $v_1 \cdot (n - 1) = k_1 \cdot (m - 1)$. При том, мора да важи да је $0 \leq v_1 < m$ и $0 \leq k_1 < n$. Решења има онолико колико постоји парова (v_1, k_1) који задовољавају дати услов (такви су сигурно парови $(0, 0)$ и $(m - 1, n - 1)$), али их има можда још. Означимо са d највећи заједнички делилац бројева $n - 1$ и $m - 1$. Нека је $n - 1 = s \cdot d$ и $m - 1 = t \cdot d$. Тада мора да важи $v_1 \cdot s \cdot d = k_1 \cdot t \cdot d$, тј. $v_1 \cdot s = k_1 \cdot t$, при чему су s и t узајамно прости бројеви. Ова једначина је задовољена ако и само ако постоји неко i такво да је $v_1 = i \cdot t$ и $k_1 = i \cdot s$. Из услова да је $0 \leq v_1 < m$ следи да је $0 \leq v_1 \leq m - 1 = t \cdot d$, што значи да је неопходно да важи да је $0 \leq i \leq d$. Слично, из $0 \leq k_1 < n$ следи да је $0 \leq k_1 \leq n - 1 = s \cdot d$, што је опет еквивалентно са $0 \leq i \leq d$. Дакле, парова (v_1, k_1) који задовољавају $v_1 \cdot (n - 1) = k_1 \cdot (m - 1)$, $0 \leq v_1 < m$ и $0 \leq k_1 < n$ једнак је броју вредности i које задовољавају $0 \leq i \leq d$, а то је тачно $d + 1$. Дакле, задатак се може решити тако што се израчуна НЗД бројева $m - 1$ и $n - 1$ и тај НЗД се увећа за један. НЗД се може израчунати Еуклидовим алгоритмом. Различите варијанте Еуклидовог алгоритма описане су у задатку **Највећи заједнички делилац**.

```
#include <iostream>  
  
using namespace std;  
  
int nzd(int a, int b) {  
    while (b != 0) {  
        int tmp = a % b;  
        a = b;  
        b = tmp;  
    }  
    return a;  
}  
  
int main() {  
    int m, n;  
    cin >> m >> n;  
    cout << nzd(m-1, n-1) + 1 << endl;  
    return 0;  
}
```

Задатак: Модуларни инверз

Миле и Тања морају да размене тајне поруке које се састоје од пуно великих бројева. Миле је желео да се додатно заштити и да те бројеве пре слања измени тако што је сваки број помножио тајним бројем a и израчунао остатак при дељењу са тајним бројем n . Пошто је добар математичар, Миле зна да ће Тања лако моћи да дешифрује поруку ако зна бројеве a и n и ако су они узајамно прости и зато их је изабрао баш на тај начин и унапред их је договорио са Тањом. Напиши програм који Тањи помаже да дешифрује бројеве које јој је Миле послао.

Улаз: Са стандардног улаза се уносе бројеви a и n ($2 \leq a, n \leq 10^9$), за које се зна да су узајамно прости. Након тога уносе се бројеви x_i ($0 \leq x_i < n$), њих највише 10, сваки у посебном реду све до краја стандардног улаза, који представљају бројеве које је Миле добио након шифровања оригиналних бројева.

Издаз: На стандардни издаз исписати оригиналне (дешифроване) бројеве, сваки у посебном реду.

Пример

Улаз	Изаз
3	2
5	4
1	1
2	3
3	
4	

Објашњење

Заиста, важи да је $(3 \cdot 2) \bmod 5 = 6 \bmod 5 = 1$, $(3 \cdot 4) \bmod 5 = 12 \bmod 5 = 2$, $(3 \cdot 1) \bmod 5 = 3 \bmod 5 = 3$ и $(3 \cdot 3) \bmod 5 = 9 \bmod 5 = 4$.

Решење

Нагласимо за почетак да је у програму потребно помножити два броја који могу бити до 10^9 и израчунати производ који може бити до 10^{18} . Зато је у програму потребно користити 64-битно записане бројеве. У језику C++ можемо користити тип `long long`.

Претрага грубом силом

За сваки учитани број x треба одредити y такав да је $a \cdot_n y = x$. Број y се, дакле, може одредити тако што се изврши дељење броја x бројем a по модулу n . Дељење по модулу је веома специфична операција, другачија од свих операција по модулу које смо до сада срели. Чак ни постојање количника по модулу n није увек гарантовано (ако су бројеви a и n узајамно прости, као што јесу у овом задатку тада модулари количник постоји за свако x и јединствен међу бројевима од 0 до $n - 1$).

Модуларно дељење грубом силом подразумева би примену алгоритма претраге за свако x тј. испробавање свих могућих бројева y од 0 до $n - 1$ и проверу да ли је $a \cdot_n y = x$. Пошто n може бити прилично велики број, проналажење инверза и за једно x може бити веома неефикасно.

```
#include <iostream>
```

```
using namespace std;
```

```
// pokusavamo da podelimo x sa a po modulu n
// ako postoji rezultat se smesta u y, a funkcija vraca podatak o tome
// da li kolicnik postoji
bool deli(long long x, long long a, long long n, long long& y) {
    // resenje grubom silom
    // posto je (a * y) mod n = (a mod n * y mod n) mod n
    // mozemo a zameniti sa a % n
    a %= n;
    // proveravamo sve moguće kandidate za y
    for (y = 0; y < n; y++)
        // ako je
        if ((a * y) % n == x)
            return true;
    // kolicnik ne postoji
    return false;
}
```

```
int main() {
    // učitavamo brojeve
    long long a, n;
    cin >> a >> n;
    // učitavamo sve brojeve do kraja ulaza
    long long x;
    while (cin >> x) {
        // delimo ih sa a po modulu n (po uslovima zadatka kolicnik
        // postoji)
        long long y;
        deli(x, a, n, y);
    }
```

```

    cout << y << endl;
}
return 0;
}

```

Уместо независног дељења сваког броја x бројем a (по модулу n), бољи приступ је израчунавање модуларног инверза броја a и касније множење (по модулу n) свих бројева x тим инверзом

За дати број a потребно је пронаћи његов инверз по модулу n тј. број a^{-1} такав да је $a^{-1} \cdot_n a = 1$, тј. да је $(a^{-1} \cdot a) \bmod n = 1$. Ако се такав број нађе, онда важи да је $a^{-1} \cdot_n (a \cdot_n y) = a^{-1} \cdot_n x$. Пошто је $a^{-1} \cdot_n (a \cdot_n y) = (a^{-1} \cdot a) \cdot_n y = 1 \cdot_n y = y$, важи да је $y = a^{-1} \cdot_n x$, што нам омогућава да дешифровање тј. дељење по модулу извршимо операцијом множења по модулу.

Одређивање модуларног инверза се може урадити грубом силом тако што се редом испитају сви бројеви a^{-1} од 0 до $n - 1$ и провери се да ли је $a^{-1} \cdot_n a = 1$. Иако се претрага овај пут спроводи само једном, а не за свако x изнова, ово решење је опет веома неефикасно у случају када је n велики број.

```

#include <iostream>

using namespace std;

// odredjujemo inverz broja a po modulu n i smestamo ga u promenljivu t
// funkcija vraca podatak o tome da li modularni inverz postoji
bool inverz(long long a, long long n, long long& t) {
    // resenje grubom silom

    // posto je (a * t) mod n = (a mod n * t mod n) mod n
    // mozemo a zameniti sa a % n
    a %= n;
    // proveravamo sve moguće kandidate za inverz
    for (t = 0; t < n; t++)
        // ako je ostatak 1, nasli smo inverz
        if ((t * a) % n == 1)
            return true;
    // inverz ne postoji
    return false;
}

int main() {
    // ucitavamo brojeve
    long long a, n;
    cin >> a >> n;
    // trazimo inverz elementa a po modulu n (po uslovima zadatka on
    // mora da postoji)
    long long inv_a;
    inverz(a, n, inv_a);
    // ucitavamo sve brojeve do kraja ulaza
    long long x;
    while (cin >> x)
        // delimo ih sa a po modulu tako sto ih mnozimo sa inverzom od a
        cout << (inv_a * x) % n << endl;
    return 0;
}

```

Проширени Еуклидов алгоритам и Безуова теорема

Једна изразито важна последица Еуклидовог алгоритма описаног у задатку [Највећи заједнички делилац](#) је Безуова теорема која каже да за свака два природна броја a и b постоје цели бројеви s и t тако да важи да је $s \cdot a + t \cdot b = \text{nzd}(a, b)$. Бројеви s и t се могу израчунати тзв. проширеним Еуклидовим алгоритмом. Посматрајмо наредну табелу.

$$\begin{array}{lll}
 r_0 & = & a \\
 r_1 & = & b \\
 r_2 & = & r_0 - q_1 r_1 \\
 \dots & & \\
 r_{i+1} & = & r_{i-1} - q_i r_i \\
 \dots & & \\
 r_k & = & r_{k-2} - q_{k-1} r_{k-1} \\
 r_{k+1} & = & 0
 \end{array}
 \qquad
 \begin{array}{lll}
 s_0 & = & 1 \\
 s_1 & = & 0 \\
 s_2 & = & s_0 - q_1 s_1 \\
 \dots & & \\
 s_{i+1} & = & s_{i-1} - q_i s_i \\
 \dots & & \\
 s_k & = & s_{k-2} - q_{k-1} s_{k-1} \\
 s_{k+1} & = & s_{k-1} - q_k s_k
 \end{array}
 \qquad
 \begin{array}{lll}
 t_0 & = & 0 \\
 t_1 & = & 1 \\
 t_2 & = & t_0 - q_1 t_1 \\
 \dots & & \\
 t_{i+1} & = & t_{i-1} - q_i t_i \\
 \dots & & \\
 t_k & = & t_{k-2} - q_{k-1} t_{k-1} \\
 t_{k+1} & = & t_{k-1} - q_k t_k
 \end{array}$$

У сваком њеном кораку важи да је $s_i \cdot a + t_i \cdot b = r_i$. Заиста, у прва два корака једнакост је тривијално испуњена. Ако претпоставимо да важи $s_{i-1}a + t_{i-1}b = r_{i-1}$ и $s_i a + t_i b = r_i$, тада је на основу табеле $r_{i+1} = (s_{i-1}a + t_{i-1}b) - q_i(s_i a + t_i b) = (s_{i-1} - q_i s_i) \cdot a + (t_{i-1} - q_i t_i) \cdot b$. Пошто је у табели стављено да је $s_{i+1} = s_{i-1} - q_i s_i$ и $t_{i+1} = t_{i-1} - q_i t_i$, и у кораку $i + 1$ важи да је $s_{i+1}a + t_{i+1}b = r_{i+1}$. Дакле, важи да је $s_k a + t_k b = r_k$, а пошто је r_k НЗД бројева a и b (што смо доказали у задатку [Највећи заједнички делилац](#), бројеви s_k и t_k су тражени Безуови коефицијенти.

Додатно, важи да је $s_{k+1} = \pm \frac{b}{nzd(a,b)}$ и $t_{k+1} = \pm \frac{a}{nzd(a,b)}$. Заиста, важи да су бројеви s_{k+1} и t_{k+1} узајамно прости (то следи из идентитета $s_k t_{k+1} - t_k s_{k+1} = (-1)^k$). Пошто уз то важи и да је $a s_{k+1} + b t_{k+1} = 0$, важи да број s_{k+1} дели број b док t_{k+1} дели број a , па тврђење следи. Може се показати и да бројеви s_i и t_i образују серије алтернирајућег знака које су (после неких првих чланова) строго растуће по апсолутној вредности. Зато важи да је $|s_i| < \frac{b}{nzd(a,b)}$ и $|t_i| < \frac{a}{nzd(a,b)}$, што је важно својство које нас осигурава од настанка прекорачења током израчунавања.

Прикажимо проширени Еуклидов алгоритам на примеру бројева 86 и 34.

$$\begin{array}{lll}
 r_0 & = & 86 \\
 r_1 & = & 34 \\
 r_2 & = & 86 - 2 \cdot 34 = 18 \\
 r_3 & = & 34 - 1 \cdot 18 = 16 \\
 r_4 & = & 18 - 1 \cdot 16 = 2 \\
 r_5 & = & 16 - 8 \cdot 2 = 0
 \end{array}
 \qquad
 \begin{array}{lll}
 s_0 & = & 1 \\
 s_1 & = & 0 \\
 s_2 & = & 1 - 2 \cdot 0 = 1 \\
 s_3 & = & 0 - 1 \cdot 1 = -1 \\
 s_4 & = & 1 - 1 \cdot (-1) = 2 \\
 s_5 & = & -1 - 8 \cdot 2 = -17
 \end{array}
 \qquad
 \begin{array}{lll}
 t_0 & = & 0 \\
 t_1 & = & 1 \\
 t_2 & = & 0 - 2 \cdot 1 = -2 \\
 t_3 & = & 1 - 1 \cdot (-2) = 3 \\
 t_4 & = & -2 - 1 \cdot 3 = -5 \\
 t_5 & = & 3 - 8 \cdot (-5) = 43
 \end{array}$$

Дакле $nzd(86, 34) = 2$, важи да је $2 \cdot 86 + (-5) \cdot 34 = 2$, и Безуови коефицијенти су $s = 2$ и $t = -5$, серија бројева s је 1, 0, 1, -1, 2, -17, а серија бројева t је 0, 1, -2, 3, -5, 43. Последњи коефицијенти у тој серији су заиста количници бројева a и b са њиховим НЗД.

Имплементација је једноставно уопштење основне имплементације Еуклидовог алгоритма. У по две променљиве одржавамо две узастопне вредности низа r_i , низа s_i и низа t_i , у петљи их ажурирамо на основу приказаних формула све док члан низа r_i не постане једнак нули.

Видећемо да се проширени Еуклидов алгоритам може применити у много различитих ситуација.

Вратимо се нашем оригиналном проблему. Задатак смо свели на одређивање модуларног инверза броја a по модулу n тј. на то да треба пронаћи број q тако да је $a^{-1} \cdot a = q \cdot n + 1$, тј. $-q \cdot n + a^{-1} \cdot a = 1$. Пошто су бројеви a и n узајамно прости по претпоставци задатка (у супротном не бисмо могли да гарантујемо постојање решења), важи да је НЗД бројева a и n једнак 1. На основу Безуове теореме могуће је пронаћи бројеве s и t такве да је $s \cdot n + t \cdot a = 1$. Када проширеним Еуклидовим алгоритмом пронађемо те коефицијенте, коефицијент t је тражени модуларни инверз. Ако је тај број негативан, онда се за инверз може узети $t + n$ (јер тада важи $(s - a) \cdot n + (t + n) \cdot a = 1$, а пошто је $|t| < \frac{n}{nzd(a,n)} = n$, број $t + n$ је сигурно позитиван).

```
#include <iostream>
```

```
using namespace std;
```

```
long long bezu(long long a, long long b, long long& s, long long& t) {
    long long r0 = a, r1 = b;
    long long s0 = 1, s1 = 0;
    long long t0 = 0, t1 = 1;
    while (r1 > 0) {
        long long q = r0 / r1;
```

```
    long long tmp;

    tmp = r0;
    r0 = r1;
    r1 = tmp - q*r1;

    tmp = s0;
    s0 = s1;
    s1 = tmp - q*s1;

    tmp = t0;
    t0 = t1;
    t1 = tmp - q*t1;
}
s = s0; t = t0;
return r0;
}

bool inverz(long long a, long long n, long long& t) {
    long long s;
    long long r = bezu(n, a, s, t);
    if (t < 0) t += n;
    return r == 1;
}

int main() {
    long long a, n;
    cin >> a >> n;
    long long inv_a;
    inverz(a, n, inv_a);
    long long x;
    while (cin >> x)
        cout << (inv_a * x) % n << endl;
    return 0;
}
```

Приметимо да нам вредност броја s заправо није потребна, па се њено израчунавање у овој специјалној примени проширеног Еуклидовог алгоритма заправо може изоставити.

```
#include <iostream>

using namespace std;

// izracunavamo broj t takav da je a * t mod n = 1
// vracamo podatak o tome da li takav broj t postoji
bool inverz(long long a, long long n, long long& t) {
    long long r = n, r1 = a;
    t = 0; long long t1 = 1;
    while (r1 > 0) {
        long long q = r / r1;
        long long tmp;

        tmp = r;
        r = r1;
        r1 = tmp - q*r1;

        tmp = t;
        t = t1;
        t1 = tmp - q*t1;
    }
}
```



```

    }
    if (t < 0) t += n;
    return r == 1;
}

int main() {
    long long a, n;
    cin >> a >> n;
    long long inv_a;
    inverz(a, n, inv_a);
    long long x;
    while (cin >> x)
        cout << (inv_a * x) % n << endl;
    return 0;
}

```

Ојлерова теорема

Још један начин да ефикасно израчунамо модуларни инверз је коришћење Ојлерове теореме. Нека $\varphi(n)$ означава број бројева који су узајамно прости са бројем n (о њој и начину њеног ефикасног израчунавања било је речи у задатку **Ојлерова функција**). Ојлерова теорема тврди да ако су бројеви a и n узајамно прости важи да је $a^{\varphi(n)} \bmod n = 1$ тј. да је $(a^{\varphi(n)-1}) \cdot_n a \bmod n = 1$. Зато је a^{-1} по модулу n једнак броју $a^{\varphi(n)-1}$. Пошто експонент може бити велики број степеновања треба извршити неким ефикасним алгоритмом степеновања (степеновање се, наравно, врши по модулу n).

Уместо доказа Ојлерове теореме дајмо само мало појашњење на једном примеру. Посматрајмо број $n = 12$ и остатке при дељењу са 12 који су узајамно прости са 12. То су бројеви 1, 5, 7 и 11 и има их тачно $\varphi(12) = 4$. Узмимо било који број који је узајамно прост са 12. Нека то буде, на пример, број $a = 7$. Помножимо по модулу 12 све наведене остатке бројем 7. Тако добијамо бројеве $1 \cdot 7 \bmod 12 = 7$, затим $5 \cdot 7 \bmod 12 = 11$, затим $7 \cdot 7 \bmod 12 = 1$ и на крају $11 \cdot 7 \bmod 12 = 5$. Приметимо да смо добили исте бројеве као на почетку (једино им се редослед променио). Кључни увид у доказу је да ће то увек бити случај када су a и n узајамно прости. Зато су производи $(7 \cdot 1) \cdot (7 \cdot 5) \cdot (7 \cdot 7) \cdot (7 \cdot 11)$ тј. $7^4 \cdot (1 \cdot 5 \cdot 7 \cdot 11)$ и $1 \cdot 5 \cdot 7 \cdot 11$ једнаки по модулу 12 и зато број 7^4 при дељењу са 12 даје остатак 1, што је управо тврђење Ојлерове теореме.

```
#include <iostream>
```

```
using namespace std;
```

```

// broj brojeva manjih od n koji su uzajamno prosti sa n
long long phi(long long n) {
    // Ojlerova formula kaze da je taj broj jednak
    // n * proizvod(1 - 1/p) za sve proste brojeve p koji dele broj n
    long long d = 2;
    long long proizvod = n;
    // dok broj n ne postane prost
    while (d*d <= n) {
        if (n % d == 0) {
            // nasli smo nov prost faktor pa azuriramo proizvod
            proizvod = (proizvod / d) * (d - 1);
            // uklanjamo ostala pojavljivanja tog faktora
            while (n % d == 0)
                n /= d;
        }
        // prelazimo na sledeceg kandidata
        d++;
    }
    // n je prost broj, pa treba azurirati proizvod i za njega
    if (n > 1)
        proizvod = (proizvod / n) * (n - 1);
}

```

```
// vracamo rezultat
return proizvod;
}

// izracunavamo x^k mod n koristeći algoritam efikasnog stepenovanja
long long stepen_po_modulu(long long x, long long k, long long n) {
    k %= n;
    long long stepen = 1;
    while (k > 0) {
        if (k % 2 == 1)
            stepen = (stepen * x) % n;
        x = (x * x) % n;
        k /= 2;
    }
    return stepen;
}

// izracunavamo broj t takav da je a * t mod n = 1
// funkcija pretpostavlja da su a i n uzajamno prosti i tada t uvek postoji
long long inverz(long long a, long long n) {
    return stepen_po_modulu(a, phi(n)-1, n);
}

int main() {
    // učitavamo brojeve
    long long a, n;
    cin >> a >> n;
    // trazimo inverz elementa a po modulu n (po uslovima zadatka on
    // mora da postoji)
    long long inv_a = inverz(a, n);
    // učitavamo sve brojeve do kraja ulaza
    long long x;
    while (cin >> x)
        // delimo ih sa a po modulu tako sto ih mnozimo sa inverzom od a
        cout << (inv_a * x) % n << endl;
    return 0;
}
```

Задатак: Кинеска теорема

Пера покушава да паралелизује свој програм који ради над x података тако да сваки процесор обрађује исти број података. Ако распореди податке на n_1 процесора, остаје му a_1 података вишка, ако распореди податке на n_2 процесора, остаје му a_2 података вишка, а ако их распореди на n_3 процесора, остаје му a_3 податка вишка. Ако се знају бројеви a_1, n_1, a_2, n_2, a_3 и n_3 и ако се зна да су бројеви n_i узајамно прости, напиши програм који одређује x .

Улаз: Са стандардног улаза уносе се бројеви a_1, n_1, a_2, n_2, a_3 и n_3 ($2 \leq n_i \leq 10^5, 0 \leq a_i < n_i$). Сваки пар се наводи у посебном реду, а бројеви су раздвојени размаком. Бројеви n_1, n_2 и n_3 су узајамно прости.

Излаз: На стандардни излаз исписати један природан број - јединствен природан број мањи од производа $n_1 \cdot n_2 \cdot n_3$ који задовољава дате услове.

Пример

Улаз	Излаз
2 3	23
3 5	
2 7	

Објашњење

Када се 23 податка подели на 3 процесора, сваки процесор добија 7 податка (укупно 21) и 2 податка остају нераспоређена. Када се подели на 5 процесора сваки процесор добија по 4 податка (укупно 20) и три податка остају нераспоређена. Када се подели на 7 процесора, сваки процесор добија по 3 податка (укупно 21) и опет 2 податка остају нераспоређена. Слично би важило и за 128 података, 233 податка итд., али 23 је једини број мањи од $3 \cdot 5 \cdot 7 = 105$ за који ово важи.

Решење

Груба сила

Решење грубом силом подразумева да линеарном претрагом пронађемо најмањи број који при дељењу са датим делиоцима даје дате остатке.

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

using namespace std;

int main(void) {
    // učitavamo ulazne podatke
    long long a[3], n[3];
    for (int i = 0; i < 3; i++)
        cin >> a[i] >> n[i];

    // rezultat izracunavamo na osnovu kineske teoreme o ostacima
    for (long long rezultat = 0; true; rezultat++)
        if (rezultat % n[0] == a[0] &&
            rezultat % n[1] == a[1] &&
            rezultat % n[2] == a[2]) {
            cout << rezultat << endl;
            break;
        }

    return 0;
}
```

Сито

Постоји један поступак који није превише математички захтеван, али који није оптималан у смислу ефикасности (но, испоставља се да је довољно ефикасан да реши задатак у оквиру постављених ограничења). Приступ је заснован на паметној претрази. Ако број r са бројем n_1 даје остатак a_1 онда је он сигурно један од чланова аритметичког низа $a_1, a_1 + n_1, a_1 + 2n_1, \dots$ У петљи редом проверавамо ове бројеве све док не наиђемо на први елемент који при дељењу са n_2 даје остатак a_2 . Када такав број a_{12} нађемо (он ће бити најмањи позитиван број са особиним да при дељењу са n_1 и n_2 даје редом остатке a_1 и a_2), знамо да ће сваки наредни број који задовољава то својство бити члан аритметичког низа $a_{12}, a_{12} + n_1 \cdot n_2, a_{12} + 2 \cdot n_1 \cdot n_2, \dots$ Редом претражујемо елементе овог низа док не наиђемо на елемент a_{123} који при дељењу са n_3 даје остатак a_3 . Пошто су у нашем задатку дата само три елемента, ово је и коначно решење. Поступак би се лако уопштио тако што би се након проналажења a_{123} посматрали бројеви $a_{123}, a_{123} + n_1 \cdot n_2 \cdot n_3, a_{123} + 2 \cdot n_1 \cdot n_2 \cdot n_3$ и тако даље.

Прикажимо претходни алгоритам на једном примеру. Нека је $(a_1, n_1) = (2, 3)$, $(a_2, n_2) = (3, 5)$ и $(a_3, n_3) = (2, 7)$. Важи да је $N = n_0 \cdot n_1 \cdot n_2 = 105$.

Посматрамо прво аритметички низ $a_1 + k \cdot n_1$ чији су чланови 2, 5, 8, 11 итд. и у њему тражимо први број који при дељењу са $n_2 = 5$ даје тражени остатак $a_2 = 3$. Први такав број је $a_{12} = 8$.

Сада посматрамо низ $a_{12} + k \cdot (n_1 \cdot n_2)$ тј. низ 8, 23, 38 итд. и у њему тражимо први елемент који при дељењу са $n_3 = 7$ даје тражени остатак 2. То је број $a_{123} = 23$, који је коначан резултат.

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

```

using namespace std;

int main() {
    // učitavamo delioce i ostatke
    int k = 3;
    vector<long long> a(k), n(k);
    for (int i = 0; i < k; i++)
        cin >> a[i] >> n[i];

    // sortiramo nizove opadajuće po n - nizovi su kratki pa koristimo
    // naivnu implementaciju selection sort algoritma
    for (int i = 0; i < k; i++)
        for (int j = i+1; j < k; j++)
            if (n[i] > n[j]) {
                swap(n[i], n[j]);
                swap(a[i], a[j]);
            }

    // "sito"
    long long m = a[0], p = n[0];
    for (int i = 1; i < k; i++) {
        while (m % n[i] != a[i])
            m += p;
        p *= n[i];
    }

    // prijavljujemo rezultat
    cout << m << endl;

    return 0;
}

```

Кинеска теорема о остацима

Иако се у задатку посматрају тачно три делиоца и остатка, размотримо мало општији случај проналажења броја r такво да за k узајамно простих бројева $n_1, \dots, n_k$ важи да је $r \bmod n_1 = a_1, \dots, r \bmod n_k = a_k$. Сличан проблем решавања система конгруенција посматран је још у древној Кини, тако да је поступак решавања који ћемо описати и применити у решењу заснован на тврђењу које се назива **кинеска теорема о остацима**. Она тврди да ако је $N = n_1 \dots n_k$ производ свих бројева n_i и ако су бројеви $n_1, \dots, n_k$ међусобно, у паровима, узајамно прости, тада задати систем услова има јединствено решење r такво да је $0 \leq r < N$.

Ако за свако i такво да је $1 \leq i \leq k$ пронађемо број r_i такав да при дељењу са бројем n_i даје остатак a_i док при дељењу са свим другим бројевима n_j за $1 \leq j \leq k$ и $j \neq i$ даје остатак 0, тада ће тражени резултат r бити једнак збиру свих бројева r_i по модулу N тј. важиће да је $r = r_1 +_N \dots +_N r_k$, односно $r = (r_1 + \dots + r_k) \bmod N$.

Рецимо и да је овај поступак веома сличан оном код конструкције Лагранжевог интерполационог полинома тј. да је Лагранжев поступак интерполације заправо специјалан случај кинеске теореме о остацима, када се уместо бројева посматра прстен полинома.

Докажимо претходно тврђење. Ако важи да је $N \bmod n_i = 0$, тада за сваки број x важи да је $(x \bmod N) \bmod n_i = x \bmod n_i$. Заиста, ако је $y = (x \bmod N) \bmod n_i$ тада постоји неки број q такав да је $x \bmod N = q \cdot n_i + y$. Међутим, тада постоји неки број q' такав да је $x = q' \cdot N + q \cdot n_i + y$. Зато на основу особина сабирања, одузимања и множења по модулу описаних у задатку **Операције по модулу**, важи да је $x \bmod n_i = ((q' \cdot N) \bmod n_i + (q \cdot n_i) \bmod n_i + (y \bmod n_i)) \bmod n_i = (y \bmod n_i) \bmod n_i = y \bmod n_i = y$ (jer је $N \bmod n_i = 0$, $n_i \bmod n_i = 0$ и $0 \leq y < n_i$ па зато и $y \bmod n_i = y$).

Зато знамо да ће за свако $1 \leq i \leq k$ важити и

$$\begin{aligned}
 r \bmod n_i &= ((r_1 + \dots + r_k) \bmod N) \bmod n_i \\
 &= (r_1 + \dots + r_k) \bmod n_i \\
 &= (r_1 \bmod n_i + \dots + r_i \bmod n_i + \dots + r_k \bmod n_i) \bmod n_i \\
 &= (0 + \dots + a_i + \dots + 0) \bmod n_i = a_i
 \end{aligned}$$

јер је $0 \leq a_i < n_i$.

Размотримо сада проблем одређивања вредности r_i . Ако је број r_i дељив са свим коефицијентима n_j осим евентуално са n_i он је дељив и са $p_i = N/n_i$ тј. мора да представља неки умножак броја p_i тј. мора бити облика $c_i \cdot p_i$, за неки коефицијент c_i . Да би r_i при дељењу са n_i дао остатак a_i желимо да важи да је $c_i \cdot n_i \cdot p_i = a_i$. Одавде се намеће да се c_i може добити неким обликом дељења по модулу n_i броја a_i бројем p_i . Разматрајмо модуларни инверз p_i^{-1} броја p_i по модулу n_i , тј. број p_i^{-1} такав да је $(p_i^{-1} \cdot p_i) \bmod n_i = 1$ (проналажење модуларног инверза описали смо у задатку **Модуларни инверз**). Модуларни инверз ће постојати јер су бројеви p_i и n_i узајамно прости. Нека је $c_i = a_i \cdot n_i \cdot p_i^{-1}$. Тада је тражени број r_i производ по модулу N бројева a_i , p_i^{-1} и p_i тј. број $r_i = (a_i \cdot n_i \cdot p_i^{-1}) \cdot N \cdot p_i$ има тражена својства.

Докажимо да број $r_i = (a_i \cdot n_i \cdot p_i^{-1}) \cdot N \cdot p_i$ има тражена својства.

Важи да је $r_i = ((a_i \cdot p_i^{-1}) \cdot p_i) \bmod N$, па за свако $1 \leq j \leq k$ (укључујући и $j = i$) важи

$$r_i \bmod n_j = (((a_i \cdot p_i^{-1}) \cdot p_i) \bmod N) \bmod n_j = ((a_i \cdot p_i^{-1}) \cdot p_i) \bmod n_j.$$

- Ако је $j = i$, на основу особина множења по модулу, важи да је $r_i \bmod n_i = (((a_i \bmod n_i) \cdot ((p_i^{-1} \cdot p_i) \bmod n_i)) \bmod n_i)$. Пошто је $0 \leq a_i < n_i$, важи да је $a_i \bmod n_i = a_i$. Такође, важи да је $(p_i^{-1} \cdot p_i) \bmod n_i = 1$, па је зато $r_i \bmod n_i = a_i$.
- Слично, за свако j такво да је $j \neq i$ и $1 \leq j \leq k$ важи да је $r_i \bmod n_j$ једнако $((a_i \bmod n_j) \cdot (p_i^{-1} \bmod n_j) \cdot (p_i \bmod n_j)) \bmod n_j$. Међутим, пошто је $p_i \bmod n_j = 0$, јер је n_j фактор броја p_i , важи да је $r_i \bmod n_j = 0$.

С обзиром на то да међурезултати могу бити велики бројеви, треба бити веома обазрив на могућност прекорачења. Узевши у обзир ограничења дата у тексту задатка (да су задата само три делиоца и три остатка, као и да су сви бројеви n_i ограничени одозго са 10^5), у имплементацији може да се користи 64-битни тип за запис целих бројева који може да чува вредности до 10^{18} (али и то само ако се формуле додатно мало трансформишу). Наиме, бројеви n_i имају ограничење 10^5 тако да њихов производ $n_1 \cdot n_2 \cdot n_3$ има ограничење 10^{15} и за његову репрезентацију никако нису довољна 32 бита, али јесте довољно 64 бита. Бројеви p_i имају горње ограничење 10^{10} , док њихов инверз по модулу n_i и број a_i имају горње ограничење 10^5 , тако да би директно израчунавање производа $a_i \cdot p_i^{-1} \cdot p_i$ проузроковало прекорачење 64-битног типа података. Чак и израчунавања остатка при дељењу са N пре и након сваке операције нас не би заштитило од прекорачења. Вредност $(a_i \cdot p_i^{-1}) \bmod N$ има горње ограничење 10^{15} (јер је N велики број), па би множење тог броја са бројем p_i чије је горње ограничење 10^{10} вероватно довело до прекорачења. Чак и да се редослед операција замени и да се број $(a_i \cdot p_i) \bmod N$ множи бројем p_i^{-1} који је доста мањи нас не би заштитило, јер би се множили бројеви чија су ограничења 10^{15} и 10^5 , чији се производ опет не може репрезентовати са 64 бита.

Безбедно решење се може добити ако се примети да је $N = p_i \cdot n_i$ и да се заправо рачуна вредност израза $(a_i \cdot p_i^{-1} \cdot p_i) \bmod (p_i \cdot n_i)$ који и у бројиоцу и у имениоцу има заједнички фактор p_i . Вредност тог израза једнака је вредности израза $p_i \cdot ((a_i \cdot p_i^{-1}) \bmod n_i)$. Ова форма нам гарантује да неће доћи до прекорачења. Наиме, бројеви a_i , p_i^{-1} и n_i су мањи од 10^5 , па је такав и $(a_i \cdot p_i^{-1}) \bmod n_i$. Пошто је p_i мањи од 10^{10} њихов је производ мањи од 10^{15} и може се представити 64-битним целим бројем.

Када постоји велика опасност од прекорачења, обично је најбезбедније решење (које додуше користи мало више операција него што је то стварно неопходно) је да се дефинишу посебне функције за сабирање и множење по модулу и да се оне позивају на месту сваког сабирања и множења у коду. Као што смо видели у овом примеру, чак ни то није гаранција и потребно је било веома пажљиво анализирати проблем да би се извела формула која гарантује да до прекорачења неће доћи.

Прикажимо претходни алгоритам на једном примеру. Нека је $(a_1, n_1) = (2, 3)$, $(a_2, n_2) = (3, 5)$ и $(a_3, n_3) = (2, 7)$. Важи да је $N = n_1 \cdot n_2 \cdot n_3 = 105$.

- Важи да је $p_1 = N/n_1 = 35$. Тражимо модуларни инверз броја $p_1 = 35$ по модулу $n_1 = 3$. Он је исти као модуларни инверз броја $p_1 \bmod 3 = 2$. Једноставно проналазимо да је то број $p_1^{-1} = 2$ (јер је

$(35 \cdot 2) \bmod 3 = (2 \cdot 2) \bmod 3 = 1$). Производ бројева $a_1 = 2$, $p_1 = 35$ и $p_1^{-1} = 2$ по модулу $N = 105$ је $r_1 = 35$ и он је такав да при дељењу са 3 даје остатак 2, а при дељењу са 5 и 7 даје остатак 0.

- Важи да је $p_2 = N/n_2 = 21$. Тражимо модуларни инверз броја $p_2 = 21$ по модулу $n_2 = 5$. Он је исти као модуларни инверз броја $p_2 \bmod 5 = 1$. Једноставно проналазимо да је то број $p_2^{-1} = 1$ (јер је $(21 \cdot 1) \bmod 5 = (1 \cdot 1) \bmod 5 = 1$). Производ бројева $a_2 = 3$, $p_2 = 21$ и $p_2^{-1} = 1$ по модулу $N = 105$ је $r_2 = 63$ и он је такав да при дељењу са 5 даје остатак 3, а при дељењу са 3 и 7 даје остатак 0.
- Важи да је $p_3 = N/n_3 = 15$. Тражимо модуларни инверз броја $p_3 = 15$ по модулу $n_3 = 7$. Он је исти као модуларни инверз броја $p_3 \bmod 7 = 1$. Једноставно проналазимо да је то број $p_3^{-1} = 1$ (јер је $(15 \cdot 1) \bmod 7 = (1 \cdot 1) \bmod 7 = 1$). Производ бројева $a_3 = 2$, $p_3 = 15$ и $p_3^{-1} = 1$ по модулу $N = 105$ је $r_3 = 30$ и он је такав да при дељењу са 7 даје остатак 2, а при дељењу са 3 и 5 даје остатак 0.

Коначно решење је збир бројева $r_1 = 35$, $r_2 = 63$ и $r_3 = 30$ по модулу 105, и он је једнак 23. Он при дељењу са 3 и 7 даје остатак 2, а при дељењу са 5 даје остатак 3.

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
long long nzd(long long a, long long b) {
    while (b > 0) {
        long long ost = a % b;
        a = b;
        b = ost;
    }
    return a;
}
```

```
// t je inverz broja a po modulu n tj. broj takav da je (a * t) mod n = 1
```

```
// funkcija vraca da li je uspela da izracuna broj t
```

```
bool inverz(long long a, long long n, long long& t) {
    long long r = n, r1 = a;
    t = 0; long long t1 = 1;
    while (r1 > 0) {
        long long q = r / r1;
        long long tmp;

        tmp = r;
        r = r1;
        r1 = tmp - q*r1;

        tmp = t;
        t = t1;
        t1 = tmp - q*t1;
    }
    if (t < 0) t += n;
    return r == 1;
}
```

```
// proizvod brojeva x i y po modulu n
```

```
long long pm(long long x, long long y, long long n) {
    return ((x % n) * (y % n)) % n;
}
```

```
// zbir brojeva x i y po modulu n
```

```
long long zm(long long x, long long y, long long n) {
    return ((x % n) + (y % n)) % n;
}
```

```

// racuna (z * x) mod (z * y)
long long pmp(long long x, long long y, long long z) {
    return z * (x % y);
}

// na osnovu kineske teoreme o ostacima se izracunava rezultat tako da
// za sve elemente nizova n i a vazi da je rezultat mod n[i] = a[i]
// funkcija vraca da li je rezultat bilo moguće naci
bool kto(long long n[], long long a[], int duzina, long long& rezultat) {
    long long N = 1;
    for (int i = 0; i < duzina; i++)
        N *= n[i];

    rezultat = 0;
    for (int i = 0; i < duzina; i++) {
        long long pi = N / n[i];
        long long pi_inv;
        if (!inverz(pi, n[i], pi_inv))
            return false;
        // rezultat = (rezultat + (a[i] * pi_inv * p_i) % N) % N
        // zbog prekoracenja moramo koristiti vezu
        // rezultat = (rezultat + ((p_i * (a[i] * pi_inv)) % (p_i * n[i]))) % N
        rezultat = zm(rezultat, pmp(pi_inv, n[i], pi), N);
    }
    return true;
}

int main(void) {
    // učitavamo ulazne podatke
    long long a[3], n[3];
    for (int i = 0; i < 3; i++)
        cin >> a[i] >> n[i];

    // rezultat izracunavamo na osnovu kineske teoreme o ostacima
    long long rezultat;
    if (!kto(n, a, 3, rezultat))
        cout << "Nisu uzajamno prosti" << endl;

    // prijavljujemo rezultat
    cout << rezultat << endl;

    return 0;
}

```

Примена Безуове теореме на две једначине

Поред општег поступка за решавање проблема за k остатака, у случају само два остатка постоји и веома сличан, али мало другачије формулисан поступак, који се заснива на томе да унемо да решимо систем од две једначине (две конгруенције), применом Безуове теореме.

За узајамно просте бројеве n_1 и n_2 , на основу Безуове теореме могу се пронаћи бројеви s и t такви да важи да је $s \cdot n_1 + t \cdot n_2 = 1$. Може се лако показати да коефицијент s представља модларни инверз броја n_1 по модулу n_2 , док коефицијент t представља модларни инверз броја n_2 по модулу n_1 . Зато број $r_{12} = a_1 \cdot t \cdot n_2 + a_2 \cdot s \cdot n_1$ тј. његов остатак по модулу $n_1 \cdot n_2$, евентуално увећан за $n_1 \cdot n_2$ ако је негативан, при дељењу са n_1 даје остатак a_1 , а при дељењу са n_2 даје остатак a_2 .

Важи да је

$$\begin{aligned}
r_{12} &= a_1 \cdot t \cdot n_2 + a_2 \cdot s \cdot n_1 \\
&= a_1 \cdot t \cdot n_2 + a_2 \cdot s \cdot n_1 + a_1 \cdot s \cdot n_1 - a_1 \cdot s \cdot n_1 \\
&= a_1 \cdot (t \cdot n_2 + s \cdot n_1) + s \cdot n_1 \cdot (a_2 - a_1) \\
&= a_1 + s \cdot n_1 \cdot (a_2 - a_1)
\end{aligned}$$

Зато је остатак при дељењу r_{12} бројем n_1 једнак a_1 .

Слично се доказује и да је $r_{12} = a_2 + t \cdot n_2 \cdot (a_1 - a_2)$ па је остатак при дељењу r_{12} бројем n_2 једнак a_2 .

Дакле, ако Кинеску теорему примењујемо на два броја, не морамо два пута независно рачунати модуларни инверз, него оба потребна коефицијента можемо добити једном применом проширеног Еуклидовог алгоритма.

Ако има више бројева на које је потребно применити Кинеску теорему, након одређивања решења за прва два, исти поступак се примењује да се одреди број који при дељењу са $n_1 \cdot n_2$ даје остатак r_{12} , а при дељењу са n_3 даје остатак a_3 . Ако је $k > 3$, поступак се наставља на исти начин, док се не добије коначан резултат.

Приметимо да се у формули $(a_1 \cdot t \cdot n_2 + a_2 \cdot s \cdot n_1) \bmod (n_1 \cdot n_2)$ множе два пута по три броја, као и да је модул $n_1 \cdot n_2$ прилично велики број, па привремени резултати могу да прекораче опсег 64-битног типа, чак иако су сви a_i и n_i 32-битни целих бројеви и ако се модули рачунају пре и након сваке операције множења. То се може предупредити ако се примети да је $(a_1 \cdot t \cdot n_2) \bmod (n_1 \cdot n_2) = n_2 \cdot ((a_1 \cdot t) \bmod n_1)$ и да је $(a_2 \cdot s \cdot n_1) \bmod (n_1 \cdot n_2) = n_1 \cdot ((a_2 \cdot s) \bmod n_2)$.

Прикажимо претходни алгоритам на једном примеру. Нека је $(a_1, n_1) = (2, 3)$, $(a_2, n_2) = (3, 5)$ и $(a_3, n_3) = (2, 7)$. Важи да је $N = n_0 \cdot n_1 \cdot n_2 = 105$.

Пронађимо прво бројеве s и t такве да је $s \cdot n_1 + t \cdot n_2 = 3s + 5t = 1$. Важи, на пример да је $s = 2$ и $t = -1$, јер је $3 \cdot 2 + 5 \cdot (-1) = 1$. Тражени број је $r_{12} = a_1 \cdot n_2 \cdot t + a_2 \cdot n_1 \cdot s = 2 \cdot 5 \cdot (-1) + 3 \cdot 3 \cdot 2 = 8$. Он при дељењу са 3 даје остатак 2, а при дељењу са 5 даје остатак 3.

У наредном кораку тражимо број такав да при дељењу са $n_1 \cdot n_2 = 15$ даје остатак $r_{12} = 8$, а при дељењу са $n_3 = 7$ даје остатак 2. Зато одређујемо бројеве s и t такве да је $15s + 7t = 1$. То могу бити бројеви $s = 1$ и $t = -2$. Тражени број је тада $8 \cdot 7 \cdot (-2) + 2 \cdot 15 \cdot 1 = -82$ тј. $-82 + 105 = 23$ (пошто је -82 негативан увећавамо га за $(n_1 n_2) n_3 = 105$).

```
#include <iostream>
```

```
using namespace std;
```

```
// proizvod brojeva x i y po modulu n
```

```
long long pm(long long x, long long y, long long n) {
    return ((x % n) * (y % n)) % n;
}
```

```
// racuna (z * x) mod (z * y)
```

```
long long pmp(long long x, long long y, long long z) {
    return z * (x % y);
}
```

```
// zbir brojeva x i y po modulu n
```

```
long long zm(long long x, long long y, long long n) {
    return ((x % n) + (y % n)) % n;
}
```

```
// izracunavanje Bezuovih koeficijenata prosirenim Euklidovim
```

```
// algoritom - nakon primene funkcije vazi:
```

```
// s*a + t*b = nzd(a, b)
```

```
// pri cemu funkcija vraca vrednost nzd(a, b)
```

```
long long bezu(long long a, long long b, long long& s, long long& t) {
    long long r0 = a, r1 = b;
    long long s0 = 1, s1 = 0;
```



```

long long t0 = 0, t1 = 1;
while (r1 > 0) {
    long long q = r0 / r1;
    long long tmp;

    tmp = r0;
    r0 = r1;
    r1 = tmp - q*r1;

    tmp = s0;
    s0 = s1;
    s1 = tmp - q*s1;

    tmp = t0;
    t0 = t1;
    t1 = tmp - q*t1;
}
s = s0; t = t0;
return r0;
}

// kineska teorema o ostacima za dva broja
long long kto(long long a1, long long n1, long long a2, long long n2) {
    long long m1, m2;
    // izracunavamo bezuove koeficijente tako da je  $m1*n1 + m2*n2 = 1$ 
    bezu(n1, n2, m1, m2);
    // resenje je  $(a1*m2*n2 + a2*m1*n1) \% (n1*n2)$ ,
    // ali zbog prekoracenja moramo transformisati formulu
    long long n = n1*n2;
    long long x = zm(pmp(pm(a1, m2, n), n1, n2),
                    pmp(pm(a2, m1, n), n2, n1), n);
    // popravak ako je resenje eventualno negativno
    if (x < 0)
        x += n;
    return x;
}

int main() {
    // ucitavamo prvi par
    long long a, n;
    cin >> a >> n;
    for (int j = 1; j < 3; j++) {
        // ucitavamo naredni par
        long long aj, nj;
        cin >> aj >> nj;
        // Kineskom teoremom izracunavamo resenje za tekuce vrednosti a, n i
        // ucitane vrednosti aj, nj i pripremamo a i n za sledecu iteraciju
        a = kto(a, n, aj, nj);
        n *= nj;
    }

    // ispisujemo resenje
    cout << a << endl;

    return 0;
}

```

Задатак: Билијар

Билијарски сто је правоугаоног облика димензије $m \times n$ и има четири рупе у ћошковима. Лоптица се удара из поља са целобројним координатама (x, y) (при чему то не може бити нека рупа), дуж линије која је паралелна или је под углом од 45° у односу на неку од ивица стола. Ако претпоставимо да лоптица не успорава своје кретање, да се од сваке се ивице одбија под углом од 45° , да је веома мала и да у рупу упада само ако су јој координате центра једнаке координати рупе, напиши програм који одређује да ли ће лоптица некада упасти у рупу и ако хоће у коју рупу ће упасти.

Улаз: Са стандардног улаза се учитава 6 целих бројева. Димензије стола m и n ($1 \leq m, n \leq 10^9$), координате почетне позиције лоптице x и y ($0 \leq x \leq m$ и $0 \leq y \leq n$), тако да те координате не одређују рупу и хоризонтална и вертикална компонента брзине лоптице v_x и v_y ($-1 \leq v_x, v_y \leq 1$).

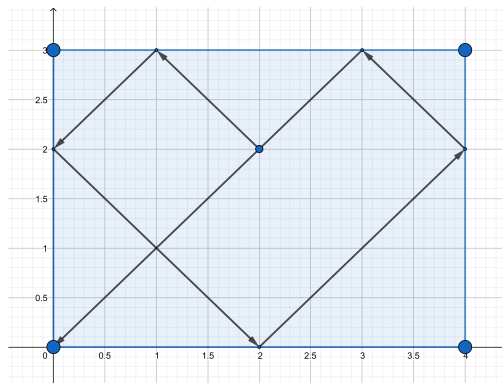
Излаз: На стандардни излаз исписати координате рупе у коју ће упасти лоптица или -1 ако ће се лоптица бесконачно дуго одбијати.

Пример 1

Улаз Излаз
4 3 2 2 -1 1 0 0

Објашњење

Путања лоптице је приказана на слици.



Слика 2.2: Билијар

Пример 2

Улаз
4 4 2 0 1 1

Излаз

-1

Пример 3

Улаз
10 10 10 1 -1 0

Излаз

-1

Решење

Случајеви када се лоптица испаљује паралелно са ивицом стола су тривијални и могу се једноставно директно разрешити. Претпоставимо зато да је лоптица испаљена дуж неке косе праве.

Основна идеја за решавање задатка је слична оној приказаној у задатку **Билијар из ћошка**. Обележимо са r_x почетно хоризонтално растојање које лоптица пређе од њеног почетног положаја до леве тј. десне ивице

стола ка којој се креће чим се испали. Растојање r_x биће једнако нули ако се лоптица већ налази на левој или десној ивици, биће једнако x ако се лоптица испаљује налево или $m - x$ ако се испаљује надесно. Током свог кретања лоптица се налази на левој или на десној ивици стола када хоризонтално пређе растојање r_x , након тога када додатно пређе једну ширину стола (када пређе $r_x + m$), након тога када додатно пређе још једну ширину стола (када пређе $r_x + 2m$) и тако даље. Дакле, лоптица ће бити на левој или десној ивици када је дужина њеног пређеног пута у хоризонталном правцу једнако $r_x + k_x m$, за неко целобројно k_x . Пошто је почетно растојање r_x до најближе ивице увек мање од ширине стола m , лоптица ће бити на левој или десној ивици ако и само ако је остатак при дељењу хоризонталног растојања са m једнак r_x . Слично, лоптица ће бити на горњој или доњој ивици стола ако и само ако је остатак при дељењу вертикалног растојања са n једнак r_y , где је r_y почетно растојање до горње или доње ивице ка којој је лоптица испаљена. Пошто је интензитет хоризонталне и вертикалне компоненте брзине увек једнак, укупно хоризонтално и вертикално пређено растојање биће увек једнако. Дакле, лоптица ће бити у рупи када хоризонтално и вертикално пређе најмање растојање r такво да је остатак при дељењу r са m једнако r_x и када је остатак при дељењу r са n једнако r_y .

Када су r_x и r_y једнаки нули (када се лоптица на почетку налази у рупи), r ће бити НЗС бројева m и n .

Када нису, онда се r може одредити применом Кинеске теореме о остацима, коју смо приказали у задатку **Кинеска теорема**. Обратимо пажњу на то да бројеви m и n не морају бити узајамно прости, тако да се не можемо директно позвати на основни облик теореме. Нека је d највећи заједнички дилац бројева m и n . Може се доказати да лопта упада у рупу ако и само ако бројеви r_x и r_y имају исти остатак при дељењу са d .

Докажимо претходно тврђење. Заиста, претпоставимо да је $r_x = q_x \cdot d + r'_x$ и да је $r_y = q_y \cdot d + r'_y$, за $0 \leq r'_x, r'_y < d$. Ако би постојао број r који би давао остатак r_x при дељењу са m и остатак r_y при дељењу са n , важило би да је $r = q_m \cdot m + r_x = q_n \cdot n + r_y$. Пошто је d дилац бројева m и n важило би да је $q_m \cdot (m' \cdot d) + q_x \cdot d + r'_x = q_n(n' \cdot d) + q_y \cdot d + r'_y$. Зато разлика $r'_x - r'_y$ мора бити дељива са d , а пошто су оба та броја мања од d , они морају бити једнаки. Дакле, ако решење постоји, остатак при дељењу бројева r_x и r_y са d мора бити исти. Доказ супротног смера је конструктиван и дат је у наставку.

Ако бројеви r_x и r_y дају исти остатак при дељењу са d , тада задатак можемо решити мало модификованом применом Кинеске теореме о остацима. Тврдимо да ће постојати јединствен број мањи од највећег заједничког садржаоца бројева m и n (који је једнак $(m \cdot n)/d$) који при дељењу са m даје остатак r_x , а при дељењу са n даје остатак r_y .

Коришћењем Безуове теореме изразимо $d = c_m \cdot m + c_n \cdot n$, за неке целобројне коефицијенте c_m и c_n . Скраћивањем са d добијамо да је $c_m \cdot (m/d) + c_n \cdot (n/d) = 1$. Тражено решење је број $(r_x \cdot c_n \cdot (n/d) + r_y \cdot c_m \cdot (m/d)) \bmod ((m \cdot n)/d)$.

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

```
using namespace std;
```

```
// zbir brojeva x i y po modulu n
```

```
long long zm(long long x, long long y, long long n) {
    return ((x % n) + (y % n)) % n;
}
```

```
// proizvod brojeva x i y po modulu n
```

```
long long pm(long long x, long long y, long long n) {
    return ((x % n) * (y % n)) % n;
}
```

```
// vrednost izraza (z * x) mod (z * y)
```

```
long long pmp(long long x, long long y, long long z) {
    return z * (x % y);
}
```

```
// izracunavanje Bezuvih koeficijenata prosirenim Euklidovim
```

```
// algoritom - nakon primene funkcije vazi:
```

```
// s*a + t*b = nzd(a, b)
```

```
// pri cemu funkcija vraca vrednost nzd(a, b)
long long bezu(long long a, long long b, long long& s, long long& t) {
    long long r0 = a, r1 = b;
    long long s0 = 1, s1 = 0;
    long long t0 = 0, t1 = 1;
    while (r1 > 0) {
        long long q = r0 / r1;
        long long tmp;

        tmp = r0;
        r0 = r1;
        r1 = tmp - q*r1;

        tmp = s0;
        s0 = s1;
        s1 = tmp - q*s1;

        tmp = t0;
        t0 = t1;
        t1 = tmp - q*t1;
    }
    s = s0; t = t0;
    return r0;
}

// kineska teorema o ostacima za dva broja
bool kto(long long a1, long long n1, long long a2, long long n2,
        long long& r) {
    long long m1, m2;
    // izracunavamo Bezuove koeficijente tako da je  $m1*n1 + m2*n2 = nzd(n1, n2)$ 
    long long d = bezu(n1, n2, m1, m2);
    // proveravamo da li resenje postoji
    if (a1 % d != a2 % d)
        return false;

    // resenje je  $(a1*m2*(n2/d) + a2*m1*(n1/d)) \% (n1*n2/d)$ ,
    // ali zbog prekoračenja sve operacije radimo po modulu
    long long n = (n1 / d) * n2;
    r = zm(pmp(pm(a1, m2, n), n1, n2 / d),
        pmp(pm(a2, m1, n), n2, n1 / d), n);

    // popravak ako je resenje eventualno negativno
    if (r < 0)
        r += n;
    return true;
}

int main() {
    int m, n, x, y, vx, vy;
    cin >> m >> n >> x >> y >> vx >> vy;
    if (vx == 0 && vy == 0)
        // loptica stoji
        cout << "-1" << endl;
    else if (vx == 0) { // loptica se kreće samo vertikalno
        if (x != 0 && x != m)
            cout << "-1" << endl;
        else if (vy > 0)
            cout << x << " " << m << endl;
    }
}
```

```

else
    cout << x << " " << 0 << endl;
} else if (vy == 0) { // loptica se krece samo horizontalno
    if (y != 0 && y != n)
        cout << "-1" << endl;
    else if (vx > 0)
        cout << m << " " << y << endl;
    else
        cout << 0 << " " << y << endl;
} else { // loptica se krece dijagonalno
    // horizontalno rastojanje potrebno da loptica dodje do leve ili
    // desne ivice
    int rx = (vx > 0 ? (m - x) : x) % m;
    // vertikalno rastojanje potrebno da loptica dodje do gornje ili
    // donje ivice
    int ry = (vy > 0 ? (n - y) : y) % n;
    // horizontalno i vertikalno rastojanje koje loptica predje dok ne
    // upadne u rupu
    long long r;
    // rastojanje odredjujemo primenom KTO
    if (!kto(rx, m, ry, n, r))
        // loptica nikada ne upada u rupu
        cout << "-1" << endl;
    else {
        // prva ivica koju loptica dodirne
        int prva_ivica_x; // 0 je leva, a 1 je desna
        if (x == 0) // vec je na levoj ivici
            prva_ivica_x = 0;
        else if (x == m) // vec je na desnoj ivici
            prva_ivica_x = 1;
        else // zavisi od brzine
            prva_ivica_x = vx < 0 ? 0 : 1;
        int prva_ivica_y; // 0 je donja, a 1 je gornja
        if (y == 0) // vec je na donjoj ivici
            prva_ivica_y = 0;
        else if (y == n) // vec je na gornjoj ivici
            prva_ivica_y = 1;
        else // zavisi od brzine
            prva_ivica_y = vy < 0 ? 0 : 1;
        // broj prelaza stola koja loptica napravi
        long long broj_prelaza_x = (r - rx) / m;
        long long broj_prelaza_y = (r - ry) / n;
        // koordinate na kojima se loptica nalazi
        int rupax = (prva_ivica_x + broj_prelaza_x) % 2 == 0 ? 0 : m;
        int rupay = (prva_ivica_y + broj_prelaza_y) % 2 == 0 ? 0 : n;
        cout << rupax << " " << rupay << endl;
    }
}
return 0;
}

```

Задатак: Ојлерова функција

Напиши програм који одређује колико има природних бројева m мањих или једнаких од датог броја n који су узајамно прости са n тј. за које је $1 \leq m \leq n$ и $\text{nzd}(m, n) = 1$.

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

Излаз: На стандардни излаз исписати само тражени број.

Пример

Улаз Излаз
9 6

Објашњење

Бројеви узајамно прости са бројем n су бројеви 1, 2, 4, 5, 7 и 8 и има их укупно 6.

Пример 2

Улаз

1

Излаз

1

Објашњење

Број 1 задовољава услове $1 \leq 1$ и $nzd(1, 1) = 1$.

Решење

Груба сила

Функцију $\varphi(n)$ која израчунава број бројева узајамно простих са n проучавао је Леонард Ојлер, па се ова функција обично назива Ојлерова функција.

Наивни приступ израчунавању заснивао би се на бројању елемената који су узајамно прости са датим бројем n (дакле, на бројању елемената који задовољавају дати услов), при чему бисмо проверу да ли су два броја узајамно проста могли засновати на израчунавању НЗД и примени Еуклидовог алгоритма који смо описали у задатку **Највећи заједнички делилац** (бројеви су узајамно прости ако и само ако им је НЗД једнак 1).

```
#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  
}
```

```
// broj brojeva iz intervala [1, n] koji su uzajamno prosti sa n
```

```
int phi(int n) {  
    int broj = 0;  
    for (int i = 1; i <= n; i++)  
        if (nzd(i, n) == 1)  
            broj++;  
    return broj;  
}
```

```
int main() {  
    // izracunavamo vrednost phi za uneti broj  
    int n;  
    cin >> n;  
    cout << phi(n) << endl;
```

```

return 0;
}

```

Растављање на просте чиниоце

Ипак, математичка својства Ојлерове функције нам дају начин да њену вредност много брже израчунамо. Функција је мултипликативна, тј. ако су a и b узајамно прости бројеви, тада је $\varphi(a \cdot b) = \varphi(a) \cdot \varphi(b)$. Вредности мултипликативних функција се могу јако ефикасно израчунати на основу разлагања аргумента на просте чиниоце и израчунавању вредности функције за просте чиниоце (ту смо технику видели, на пример, у задатку **Савршени бројеви**). Ако је p прост број, тада је $\varphi(p) = p - 1$ (јер су сви бројеви мањи од p узајамно прости са p). Број узајамно простих бројева са бројем p^k је $\varphi(p^k) = p^k - p^{k-1} = p^k \cdot \left(\frac{p-1}{p}\right)$. Од свих p^k бројева из интервала $[1, p^k]$ једини који нису узајамно прости са p^k су умношци броја p (то су бројеви $p, 2p, 3p, \dots, p^{k-1} \cdot p$). Њих је p^{k-1} , што доказује тврђење. Дакле, ако се број n раставља као $p_1^{k_1} \cdot \dots \cdot p_m^{k_m}$, тада је $\varphi(n) = \left(p_1^{k_1} \cdot \left(\frac{p_1-1}{p_1}\right)\right) \cdot \dots \cdot \left(p_m^{k_m} \cdot \left(\frac{p_m-1}{p_m}\right)\right)$ тј. $(p_1^{k_1} \cdot \dots \cdot p_m^{k_m}) \cdot \frac{(1-p_1) \cdot \dots \cdot (1-p_m)}{p_1 \cdot \dots \cdot p_m}$ односно $n \cdot \frac{(1-p_1) \cdot \dots \cdot (1-p_m)}{p_1 \cdot \dots \cdot p_m}$.

Ово нам даје начин за ефикасно израчунавање вредности $\varphi(n)$. На основу алгоритма датог у задатку **Растављање на просте чиниоце** и сваки пут када наиђемо на прост чинилац p_i производ (који иницијализујемо на 1) množимо вредношћу $\frac{p_i-1}{p_i}$. Након тога из броја уклањамо сва појављивања чиниоца p_i (за свако појављивање чиниоца, без обзира на његову вишеструкост, производ се множи само једним чиниоцем).

На крају као резултат враћамо акумулирани производ помножен вредношћу броја n . Пошто се n мења током одређивања његових простих чинилаца, боље решење је производ на почетку иницијализовати на n .

```

#include <iostream>

using namespace std;

// број бројева мањих од n који су узајамно прости са n
int phi(int n) {
    // Ојлерова формула каже да је тај број једнак
    // n * производ(1 - 1/p) за све просте бројеве p који деле број n
    int d = 2;
    int proizvod = n;
    // док број n не постане прост
    while (d*d <= n) {
        if (n % d == 0) {
            // нашли смо нов прост фактор па азурирамо производ
            proizvod = (proizvod / d) * (d - 1);
            // уклањамо остала појављивања тог фактора
            while (n % d == 0)
                n /= d;
        }
        // прелазимо на следећег кандидата
        d++;
    }
    // n је прост број, па треба азурирати производ и за њега
    if (n > 1)
        proizvod = (proizvod / n) * (n - 1);
    // враћамо резултат
    return proizvod;
}

int main() {
    int n;
    cin >> n;
    cout << phi(n) << endl;
    return 0;
}

```

2.3 Полиноми

Задатак: Вредност полинома

Са стандардног улаза се уносе степен n и реални коефицијенти полинома $y = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \dots + x \cdot a_1 + a_0$. Напиши програм који израчунава вредност тог полинома у k равномерно распоређених тачака интервала $[p, q]$.

Улаз: У првој линији стандардног улаза унети n ($2 \leq n \leq 9$) - степен полинома, у следећих $n + 1$ линија реалне вредности коефицијената полинома (редом бројеви $a_n, a_{n-1}, \dots, a_1, a_0$), затим, у наредној линији k ($2 \leq k \leq 40$) - број равномерно распоређених тачака на интервалу $[p, q]$, у наредној линији реалну вредност p - почетак интервала, и у наредној линији реалну вредност q - крај интервала.

Издаз: У k линија исписати вредност полинома у равномерно распоређеним тачакама интервала $[p, q]$ заокружену на две децимале.

Пример

Улаз	Издаз
2	4.00
1.0	9.00
2.0	16.00
1.0	25.00
7	36.00
1.0	49.00
7.0	64.00

Решење

Хорнерова шема

Вредност полинома

$$y = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \dots + x \cdot a_1 + a_0$$

се може израчунати без коришћења операције степеновања ако се полином представи у Хорнеровом облику (који смо описали, на пример, у задатку **Јарди, стопе и инчи** или **UNIX време**):

$$y = (((\dots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + \dots + a_1) \cdot x + a_0$$

Ако се y иницијализује са 0, у $n + 1$ итерација се израчунава $y = y \cdot x + a_n, y = y \cdot x + a_{n-1}, \dots, y = y \cdot x + a_0$.

Генерисање k равномерних тачака у интервалу $[p, q]$ описали смо у задатку **Подела интервала на једнаке делове**. Да би се вредност полинома израчунала у k равномерно распоређених тачака интервала $[p, q]$ - вредност полинома се израчунава у тачкама $p + i \cdot h$ за $i = 0, 1, \dots, k - 1$ и $h = (q - p) / (k - 1)$.

```
double vrednostPolinoma(double a[], int n, double x) {
    // Hornerova shema
    double y = 0.0;
    for (int i = n; i >= 0; i--)
        y = y*x + a[i];
    return y;
}

int main() {
    // učitavamo stepen n i koeficijente polinoma a
    // broj tacaka k i granice intervala p, q
    // ...
    double h = (q - p) / (k - 1); // duzina intervala
    double x; // tekuca tacka
    int i;
    for (i = 0, x = p; i < k; i++, x += h)
```



```

    cout << fixed << showpoint << setprecision(2)
         << vrednostPolinoma(a, n, x) << endl;
}

```

Класична дефиниција

Уместо Хорнерове шеме може се употребити и класична дефиниција вредности. Тада се у сваком кораку израчунава степен x^i . У језику C++ степен се може израчунати функцијом `pow` декларисаном у заглављу `<cmath>`. Израчунати степен се множи са коефицијентом a_i (који се налази у низу коефицијената `a` на позицији `i`) и додаје на текући збир (који је иницијално постављен на нулу). Једна могућа оптимизација овог поступка (која избегава употребу степеновања) је да се одржава вредност степена (која се иницијализује на 1) и да се у сваком кораку, након увећања збира вредност степена помножи са x (слично како је то приказано у задацима [Цифре сдесна](#) и [Број формиран од датих цифара здесна на лево](#)). Ипак, Хорнерова шема је најелегантније и најефикасније решење.

```

double vrednostPolinoma(double a[], int n, double x) {
    // klasicna definicija
    double y = 0.0;
    for (int i = 0; i <= n; i++)
        y += a[i] * pow(x, i);
    return y;
}

```

Задатак: Аритметика над полиномима

Дата су два полинома, P и Q степенима и низовима својих коефицијената. Одредити њихов збир и производ.

Улаз: У првој линији стандардног улаза налази се степен n ($0 \leq n \leq 20$) првог полинома, а у следећих $n + 1$ линија реални коефицијенти првог полинома и то редом почев од коефицијента уз највећи степен. Затим се на стандардном улазу налази степен m ($0 \leq m \leq 20$) другог полинома, а у следећих $m + 1$ линија реални коефицијенти другог полинома и то редом почев од коефицијента уз највећи степен.

Излаз: Приказати редом збир и производ, сваки у посебној линији. За сваки полином приказати у једној линији његове коефицијенте, на две децимале и то редом почев од коефицијента уз највећи степен.

Пример

Улаз	Излаз
2	2.00 2.00 1.00
2	2.00 -1.00 1.00 -2.00
1	
2	
1	
1	
-1	

Решење

Нека су дати полиноми

$$\begin{aligned}
 P_n(x) &= p_n \cdot x^n + p_{n-1} \cdot x^{n-1} + \dots + x \cdot p_1 + p_0 \\
 Q_m(x) &= q_m \cdot x^m + q_{m-1} \cdot x^{m-1} + \dots + x \cdot q_1 + q_0
 \end{aligned}$$

Збир полинома одређујемо на следећи начин:

- ако су полиноми истог степена ($n = m$) резултат је полином степена n и његови коефицијенти су једнаки збиру одговарајућих коефицијената полинома P и Q ,
- ако су полиноми различитих степени нпр. $n > m$ онда је збир полином степена n и коефицијенти од n до $m - 1$ једнаки су коефицијентима полинома P , а коефицијенти од m до 0 једнаки су збиру одговарајућих коефицијената полинома P и Q .

Производ полинома P и Q је полином R степена $n + m$. Производ одређујемо тако што множимо сваки моном једног полинома сваким мономом другог полинома и сабирамо добијене мономе истог степена.

$$P \cdot Q = \sum_{i=0}^n \sum_{j=0}^m p_i \cdot q_j \cdot x^{i+j}$$

Сабирање вршимо поступно - на почетку све коефицијенте резултујућег полинома c_k постављамо на нулу, а затим за свако i од 0 до n и за свако j од 0 до m увећамо коефицијент c_{i+j} за вредност производа $p_i \cdot q_j$ (приликом множења монома $p_i \cdot x^i$ мономом $q_j \cdot x^j$ добијамо моном $p_i \cdot q_j \cdot x^{i+j}$, који се додаје моному $c_{i+j}x^{i+j}$ тренутно акумулираном у резулату).

У имплементацији коефицијенте полинома и степен полинома можемо чувати обједињене унутар структуре.

```
struct Polinom {
    int n; // stepen
    vector<double> a; // koeficijenti an, ..., a0
};

// pravi nula polinom stepena n
Polinom napravi(int n) {
    Polinom P;
    P.n = n;
    P.a.resize(n+1, 0);
    return P;
}

// cita stepen i koeficijente sa standardnog ulaza
Polinom unos() {
    int n;
    cin >> n;
    Polinom P = napravi(n);
    for (int i = P.n; i >= 0; i--)
        cin >> P.a[i];
    return P;
}

// ispisuje koeficijente na standardni izlaz
void prikaz(Polinom P) {
    for (int i = P.n; i >= 0; i--)
        cout << fixed << setprecision(2) << showpoint << P.a[i] << " ";
    cout << endl;
}

// sabira polinome P i Q
Polinom zbir(Polinom P, Polinom Q) {
    Polinom R = napravi(max(P.n, Q.n));
    for (int i = 0; i <= R.n; i++) {
        if (i <= P.n)
            R.a[i] += P.a[i];
        if (i <= Q.n)
            R.a[i] += Q.a[i];
    }
    return R;
}

// mnozi polinome P i Q
Polinom proizvod(Polinom P, Polinom Q) {
    Polinom R = napravi(P.n + Q.n);
    for (int i = 0; i <= P.n; i++)
        for (int j = 0; j <= Q.n; j++)
            R.a[i + j] += P.a[i] * Q.a[j];
}
```

```

    return R;
}

int main() {
    Polinom P, Q, R, S;
    P = unos();
    Q = unos();
    R = zbir(P, Q);
    prikaz(R);
    R = proizvod(P, Q);
    prikaz(R);
    return 0;
}

```

Задатак: Множење полинома

Напиши програм који ефикасно одређује производ два полинома.

Улаз: Са стандардног улаза се носе два полинома. За сваки полином је у једној линији дат степен n (цео број између 1 и 50000), а затим у наредној линији коефицијенти (реални бројеви заокружени на једну децималу, раздвојени размацима). Коефицијенти се задају редом, кренувши од слободног члана тј. коефицијента уз x^0 , па закључно са коефицијентом уз x^n .

Излаз: На стандардни излаз исписати полином производ, у истом формату у ком су задати и чиниоци, осим што сви коефицијенти треба да буду заокружени на две децимале.

Пример

Улаз	Излаз
2	3
1.0 2.0 3.0	1.00 4.00 7.00 6.00
1	
1.0 2.0	

Решење

Множење можемо извршити класичним алгоритмом множења, који смо приказали у задатку [Аритметика над полиномима](#). Сложеност тог алгоритма је $O(n_1 \cdot n_2)$, где су n_1 и n_2 степени полинома који се множе.

```

#include <iostream>
#include <iomanip>
#include <vector>

using namespace std;

// funkcija mnozi dva polinoma p1*p2
vector<double> proizvod(const vector<double>& p1,
                       const vector<double>& p2) {
    int n1 = p1.size(), n2 = p2.size();
    vector<double> proizvod(n1+n2-1, 0);
    for (int i = 0; i < n1; i++)
        for (int j = 0; j < n2; j++)
            proizvod[i+j] += p1[i] * p2[j];
    return proizvod;
}

vector<double> učitajPolinom() {
    int n;
    cin >> n;
    vector<double> p(n+1, 0.0);
    for (int i = 0; i <= n; i++)
        cin >> p[i];
    return p;
}

```

```

}

int main() {
    vector<double> p1 = ucitajPolinom();
    vector<double> p2 = ucitajPolinom();
    vector<double> r = proizvod(p1, p2);
    cout << r.size() - 1 << endl;
    for (size_t i = 0; i < r.size(); i++)
        cout << fixed << showpoint << setprecision(2) << r[i] << " ";
    cout << endl;
    return 0;
}

```

Иако се неко време сматрало да је доња граница сложености множења полинома $O(n_1 \cdot n_2)$, Анатолиј Карацуба је 1960. показао да је декомпозицијом могуће добити ефикаснији алгоритам. Претпоставимо да је потребно помножити полиноме $a + bx$ и $c + dx$. Директан приступ подразумева израчунавање $ac + (ad + bc)x + bd$, што подразумева 4 множења. Карацубина кључна опаска је да се исто може остварити само са три множења (на рачун мало већег броја сабирања тј. одузимања, што није критично, јер са сабирање и одузимање обично врши брже него множење, а што важи и за полиноме, јер је сабирање и одузимање полинома операција линеарне сложености). Наиме, важи да је $ad + bc = (a + b)(c + d) - (ac + bd)$. Потребно је, дакле, само израчунати производе ac , bd и $(a + b)(c + d)$, а онда прва два производа употребити по два пута (они су потребни и директно и за израчунавање производа $ad + bc$).

Имајући овај Карацубин “трик” у виду, лако можемо направити алгоритам заснован на декомпозицији. Да би имплементација била једноставнија пре множења полиноме допуњујемо нулама тако да оба имају 2^n коефицијената.

Пошто множење задовољава једначину $T(n) = 3T(n/2) + O(n)$, Сложеност множења је онда $O(n^{\log_2 3})$.

Имплементацију најједноставније можемо направити тако да се у сваком рекурзивном позиву сви међурезултати, као и крајњи резултат смештају у посебним векторима. Међутим, таква имплементација је неефикасна и тестови показују да не доприноси побољшању ефикасности наивне процедуре. Кључни проблем је то што се током рекурзије граде вектори у којима се чувају привремени резултати и те алокације и деалокације троше јако пуно времена.

```

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

using namespace std;

// funkcija mnozi dva polinoma p1*p2 sa po 2^k koeficijenata
vector<double> karacuba(const vector<double>& p1,
                      const vector<double>& p2) {
    // broj koeficijenata polinoma
    int n = p1.size();

    // polinome stepena 0 direktno mnozimo
    if (n == 1)
        return vector<double>{p1[0] * p2[0], 0.0};

    // delimo p1 na dve polovine: a i b
    vector<double> a(n / 2), b(n / 2);
    copy_n(begin(p1), n/2, begin(a));
    copy_n(next(begin(p1)), n/2, n/2, begin(b));

    // delimo p2 na dve polovine: c i d
    vector<double> c(n / 2), d(n / 2);
    copy_n(begin(p2), n/2, begin(c));
    copy_n(next(begin(p2)), n/2, n/2, begin(d));

```

```

// Važi:
//  $(ax+b)(cx+d) = a*c*x^2 + ((a+b)*(c+d) - a*c - b*d)*x + b*d$ 

// rekurzivno računamo  $a*c$  i  $b*d$ 
vector<double> ac = karacuba(a, c);
vector<double> bd = karacuba(b, d);

// izračunavamo  $a+b$  (rezultat smeštamo u vektor a)
for (int i = 0; i < n/2; i++)
    a[i] += b[i];
// izračunavamo  $c+d$  (rezultat smeštamo u vektor c)
for (int i = 0; i < n/2; i++)
    c[i] += d[i];

// izračunavamo  $(a+b)*(c+d)$ 
vector<double> adbc = karacuba(a, c);
// izračunavamo  $(a+b)*(c+d) - a*c - b*d$ 
for (int i = 0; i < n; i++)
    adbc[i] -= ac[i] + bd[i];

// sklapamo proizvod iz delova
vector<double> proizvod(2*n, 0.0);
for (int i = 0; i < n; i++) {
    proizvod[n + i] += bd[i];
    proizvod[n/2 + i] += adbc[i];
    proizvod[i] += ac[i];
}

// vraćamo rezultat
return proizvod;
}

// najmanji broj oblika  $2^k$  koji je veci ili jednak od n
int stepenDvojke(int n) {
    int s = 1;
    while (s < n)
        s <= 1;
    return s;
}

vector<double> ucitajPolinom() {
    int n;
    cin >> n;
    vector<double> p(n+1, 0.0);
    for (int i = 0; i <= n; i++)
        cin >> p[i];
    return p;
}

int main() {
    vector<double> p1 = ucitajPolinom();
    int n1 = p1.size() - 1;
    vector<double> p2 = ucitajPolinom();
    int n2 = p2.size() - 1;
    int s1 = stepenDvojke(p1.size());
    int s2 = stepenDvojke(p2.size());
    int s = max(s1, s2);

```

```

p1.resize(s, 0.0); p2.resize(s, 0.0);

vector<double> r = karacuba(p1, p2);
cout << n1 + n2 << endl;
for (int i = 0; i <= n1 + n2; i++)
    cout << fixed << showpoint << setprecision(2) << r[i] << " ";
cout << endl;
return 0;
}

```

Пажљивија анализа показује да је могуће сву помоћну меморију алоцирати само једном и онда током рекурзије користити стално исти помоћни меморијски простор. Величина потребне помоћне меморије је $4n$ (два пута по n да се сместе полиноми $a + b$ и $c + d$ и још $2n$ да се смести њихов производ). Додатна оптимизација је да се примети да је за мале степене полинома класичан алгоритам бржи него алгоритам заснован на декомпозицији (ово је чест случај код алгоритама заснованих на декомпозицији). Експерименталном анализом се утврђује да се више исплати применити класичан алгоритам кад год је $n \leq 4$.

```

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

using namespace std;

// množimo polinome čiji su koeficijenti smešteni u vektorima
// p1[start1, start1+n) i p2[start2, start2+n)
// i rezultat smeštamo u vektor
// proizvod[start_proizvod, start_proizvod + 2n),
// koristeći pomocni memorijski prostor u vektoru
// pom[start_pom, start_pom + 4n)
void karacuba(int n,
              const vector<double>& p1, int start1,
              const vector<double>& p2, int start2,
              vector<double>& proizvod, int start_proizvod,
              vector<double>& pom, int start_pom) {

    // izlaz iz rekurzije
    if (n <= 4) {
        // klasični algoritam množenja
        for (int i = 0; i < 2*n; i++)
            proizvod[start_proizvod + i] = 0;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                proizvod[start_proizvod + i+j] +=
                    p1[start1 + i] * p2[start2 + j];
        return;
    }

    // Važi: (a+bx)*(c+dx) =
    //          a*c + ((a+b)*(c+d) - a*c - b*d)*x + b*d*x^2

    // Izračunavamo rekurzivno a*c i smeštamo ga u levu polovinu
    // proizvoda
    karacuba(n / 2, p1, start1, p2, start2,
             proizvod, start_proizvod, pom, start_pom);
    // Izračunavamo rekurzivno b*d i smeštamo ga u desnu polovinu
    // proizvoda
    karacuba(n / 2, p1, start1 + n/2, p2, start2 + n/2,
             proizvod, start_proizvod + n, pom, start_pom);
}

```

```

// Izračunavamo a+b i smeštamo ga u pomoćni vektor (na početak)
for (int i = 0; i < n/2; i++)
    pom[start_pom + i] =
        p1[start1 + i] + p1[start1 + n/2 + i];
// Izračunavamo c+d i smeštamo ga u pomoćni vektor (iza (a+b))
for (int i = 0; i < n/2; i++)
    pom[start_pom + n / 2 + i] =
        p2[start2 + i] + p2[start2 + n/2 + i];

// Rekurzivno izračunavamo (a+b)*(c+d) i smeštamo ga
// u pomoćni vektor, iza (a+b) i (c+d)
karacuba(n / 2, pom, start_pom, pom, start_pom + n / 2,
        pom, start_pom + n, pom, start_pom + 2*n);

// Izračunavamo (a+b)*(c+d) - (ac + bd)
for (int i = 0; i < n; i++)
    pom[start_pom + n + i] -=
        proizvod[start_proizvod + i] + proizvod[start_proizvod + n + i];

// Dodajemo ad+bc na sredinu proizvoda
for (int i = 0; i < n; i++)
    proizvod[start_proizvod + n/2 + i] += pom[start_pom + n + i];
}

// funkcija množi dva polinoma p1*p2 sa po 2^k koeficijenata
vector<double> karacuba(const vector<double>& p1,
        const vector<double>& p2) {
    int n = p1.size();
    // koeficijenti proizvoda
    vector<double> proizvod(2 * n);
    // pomoćni memorijski prostor potreban za realizaciju algoritma
    vector<double> pom(4 * n);
    // vršimo množenje
    karacuba(n, p1, 0, p2, 0, proizvod, 0, pom, 0);
    // vraćamo proizvod
    return proizvod;
}

// najmanji broj oblika 2^k koji je veci ili jednak od n
int stepenDvojke(int n) {
    int s = 1;
    while (s < n)
        s <= 1;
    return s;
}

vector<double> ucitajPolinom() {
    int n;
    cin >> n;
    vector<double> p(n+1, 0.0);
    for (int i = 0; i <= n; i++)
        cin >> p[i];
    return p;
}

int main() {
    vector<double> p1 = ucitajPolinom();

```

```
int n1 = p1.size() - 1;
vector<double> p2 = ucitajPolinom();
int n2 = p2.size() - 1;
int s1 = stepenDvojke(p1.size());
int s2 = stepenDvojke(p2.size());
int s = max(s1, s2);
p1.resize(s); p2.resize(s);
vector<double> r = karacuba(p1, p2);
cout << n1 + n2 << endl;
for (int i = 0; i <= n1 + n2; i++)
    cout << fixed << showpoint << setprecision(2) << r[i] << " ";
cout << endl;
return 0;
}
```


Глава 3

Алгоритми текста

3.1 Тражење узорка у тексту

Један од најчешће сретаних задатака у раду са текстом је то да се провери да ли једна ниска садржи другу.

Задатак: Префикс суфикс

Домине се слажу једна уз другу, тако што се поља на доминама постављеним једну уз другу морају поклапати. Домине обично имају само два поља, међутим, наше су домине специјалне и имају више различитих поља (означених словима). Ако све домине које слажемо имају исту шару, напиши програм који одређује како је домине могуће сложити тако да заузму што мање простора по дужини (свака наредна домина мора бити смакнута бар за једно поље). На пример,

```
ababcabab
  ababcabab
    ababcabab
```

Улаз: Први ред стандардног улаза садржи ниску малих слова енглеске абееде које представљају шаре на доминама. Дужина ниске је између 2 и 50000 карактера. У наредном реду се налази цео број n ($1 \leq n \leq 1000$) који представља број домина.

Излаз: На стандардни излаз исписати цео број који представља укупну дужину сложених домина.

Пример 1		Пример 2		Пример 3	
Улаз	Излаз	Улаз	Излаз	Улаз	Излаз
ababcabab	19	abc	15	aa	11
3		5		10	

Решење

Кључ решења овог задатка је одређивање најдужег правог префикса ниске који је уједно и њен прави суфикс (префикс и суфикс су прави, ако нису једнаки целој ниски). Такав префикс тј. суфикс зваћемо *префикс-суфикс*. Један начин да се он пронађе је груба сила (да се упореде сви прави префикси са одговарајућим суфиксима и да се од оних који се поклапају одабере најдужи).

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

```
using namespace std;
```

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

```

int maxD = 0;
for (int d = 1; d < str.size(); d++) {
    bool prefiks_suffix = true;
    for (int i = 0, j = str.size() - d; i < d; i++, j++)
        if (str[i] != str[j]) {
            prefiks_suffix = false;
            break;
        }
    if (prefiks_suffix)
        maxD = d;
}

cout << maxD + n * (str.size() - maxD) << endl;
return 0;
}

```

Ипак, много ефикасније решење засновано је на динамичком програмирању које чини основу чувеног Кнут-Морис-Пратовог (КМП) алгоритма.

Рекурзивна формулација

Задатак има рекурзивно решење. Ако је ниска s празна она нема префикс-суфикс (јер нема правих префикса ни суфикса), а ако је једнословна, тада јој је једини префикс-суфикс празна ниска. Зато претпоставимо да је $|s| > 1$ и да је $s = s'a$ за неку ниску s' и неки карактер a . Кључни увид је то да се сваки префикс-суфикс речи s , осим празног, добија проширивањем неког префикс-суфикса речи s' за слово a . Претпоставимо да је x најдужи префикс-суфикс речи s' (он се може израчунати рекурзивно, јер је реч s' краћа од речи s) и да му је дужина d . Тада постоје ниске u и v такве да је $s' = xu = vx$. Оне су непразне јер је x прави префикс тј. прави суфикс ниске s' , па није једнак s' . Нека је b прво слово ниске u и нека је $u = bu'$, за неку реч u' . Тада је $s = xbu'a = vxa$. Ако је $a = b$, тада је $xa = xb$ најдужи префикс-суфикс ниске s и његова дужина је $d + 1$. Остаје питање шта радити у случају када је $a \neq b$. Ако је x празан, тј. ако је $d = 0$, тада је једини префикс-суфикс речи s празан. У супротном, морамо размотрити неки краћи префикс-суфикс ниске s' . Следећи кандидат је најдужи прави суфикс од x који је уједно прави префикс од x иза њега треба проверити да ли се можда може продужити словом a . Овим смо проблем свели на проналажење најдужег префикс-суфикса ниске x , а то се може решити рекурзивно (јер је реч x краћа од речи s).

Поступак испитивања кандидата креће од најдужег префикс-суфикса речи s' и наставља скраћивањем кандидата се све док се не први пут не наиђе на префикс-суфикс који се може наставити или док се не утврди да ни празан префикс-суфикс не може да се продужи. Претпоставимо да је $|s| = n$. Тада је $a = s_{n-1}$, а пошто је $|x| = d$, важи да је $b = s_d$, па се поређење $a = b$ своди на поређење $s_{n-1} = s_d$. Ако су једнаки, дужина је $d + 1$, а у супротном, ажурирамо d на дужину најдужег префикс-суфикса ниске x (тј. првих d карактера ниске s) и понављамо поређење s_d и s_{n-1} и поступак настављамо све док се не испостави да су они једнаки или да су различити и да је дужина $d = 0$.

```

#include <iostream>
#include <string>
#include <vector>

using namespace std;

// одређује најдужи прави префикс-суфикс речи str[0, ..., i]
int prefiks_sufiks(const string& str, int i);

int prefiks_sufiks_(const string& str, int k, int i) {
    int ps = prefiks_sufiks(str, k);
    if (str[ps] == str[i])
        return ps + 1;
    else if (ps == 0)
        return 0;
    return prefiks_sufiks_(str, ps - 1, i);
}

```

```

int prefiks_sufiks(const string& str, int i) {
    if (i == 0)
        return 0;
    return prefiks_sufiks_(str, i - 1, i);
}

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

    int ps = prefiks_sufiks(str, str.size() - 1);
    cout << ps + n * (str.size() - ps) << endl;
    return 0;
}

```

Динамичко програмирање

Можемо приметити да ће се у претходно описаном рекурзивном поступку многи рекурзивни позиви поклапати (више пута ће се одређивати дужина најдужег префикс-суфикса за исти префикс ниске s). Стога је потребно применити мемоизацију, или још боље динамичко програмирање. У тој ситуацији одржавамо низ d_i који садржи дужине најдужих префикс-суфикса за префикс ниске s дужине i . Тако ће d_n садржати тражену дужину најдужег префикс-суфикса целе ниске s . Прва два елемента низа можемо иницијализовати на нулу (јер празна и једнословна ниска не могу имати непразан префикс-суфикс). Могуће је чак први елемент поставити на -1 , како би се нагласило да ни празна ниска није прави суфикс-префикс празне ниске. Након тога, попуњавамо остале елементе низа, један по један.

Дужину најдужег префикс-суфикса речи $s_0 \dots s_i$ бележимо као d_{i+1} (јер та реч има тачно $i + 1$ слово). Ту дужину одређујемо тако што поредимо s_i са s_{d_i} (заиста, d_i је дужина најдужег префикс-суфикса за $s_0 \dots s_{i-1}$, па се први следећи карактер иза њега налази на позицији s_{d_i}). Ако су једнаки, важи да је $d_{i+1} = d_i + 1$. У супротном, ако је d_i једнак 0, закључујемо да је $d_{i+1} = 0$, а ако није, настављамо исти поступак тако што уместо d_i посматрамо d_{d_i} и поредимо s_i са $s_{d_{d_i}}$. Заправо, можемо променљиву d иницијализовати на d_i , а затим поредити s_i са s_d , ако су једнаки постављати d_{i+1} на $d + 1$ и прекидати поступак, а ако нису, онда постављати d_{i+1} на нулу и прекидати поступак

```

#include <iostream>
#include <string>
#include <vector>

using namespace std;

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

    // kmp[i] - je duzina najduzeg prefiks-sufiksa dela str[0,..,i]
    vector<int> kmp(str.size());
    kmp[0] = 0;
    for (int i = 1; i < str.size(); i++) {
        int k = kmp[i - 1];
        while (true) {
            if (str[i] == str[k]) {
                kmp[i] = k + 1; break;
            } else if (k == 0) {
                kmp[i] = 0; break;
            } else
                k = kmp[k - 1];
        }
    }
}

```

```
    }  
}  
  
int ps = kmp[str.size() - 1];  
cout << ps + n * (str.size() - ps) << endl;  
return 0;  
}
```

Имплементацију истог алгоритма можемо направити и мало другачије.

```
#include <iostream>  
#include <string>  
#include <vector>  
  
using namespace std;  
  
int main() {  
    string str;  
    cin >> str;  
    int n;  
    cin >> n;  
  
    vector<int> kmp(str.size() + 1);  
    kmp[0] = -1;  
    for (int i = 0; i < str.size(); i++) {  
        int k = i;  
        while (k > 0 && str[i] != str[kmp[k]])  
            k = kmp[k];  
        kmp[i + 1] = kmp[k] + 1;  
    }  
  
    cout << kmp[str.size()] + n * (str.size() - kmp[str.size()]) << endl;  
  
    return 0;  
}
```

Задатак: Подниска

Написати програм којим се за две дате речи проверава да ли је друга садржана у првој, ако јесте одредити прву позицију на којој се друга реч појављује у првој.

Улаз: На стандардном улазу налазе се две речи свака у посебној линији. Свака реч има највише 20 карактера.

Изаз: На стандардном излазу приказати прву позицију (позиције су нумерисане од 0) на којој се налази друга реч у првој, ако се друга реч не налази у првој приказати -1.

Пример 1

Улаз	Изаз
banana	1
ana	

Пример 2

Улаз	Изаз
branka	-1
ana	

Решење

Библиотечке функције претраге

Када смо речи у језику C++ читали у две променљиве типа `string` (назовимо их `igla` и `plast`, јер је тако одмах јасно шта у чему тражимо) онда позицију првог појављивања речи `igla` у речи `plast` можемо одредити помоћу `plast.find(igla)`. Ако реч `plast` не садржи реч `igla` онда `find` враћа специјалну вредност `string::npos`.

```
// učitavamo reci  
string plast, igla;  
cin >> plast >> igla;
```

```
// pretragu podniske vrsimo koriscenjem bibliotecke metode
size_t poz = plast.find(igla);
// ako niska nije nadjena, rezultat je string::npos
if (poz != string::npos)
    cout << poz << endl;
else
    cout << -1 << endl;
```

Имплементација претраге грубом силом

Осим коришћења библиотечких функција, решење је било могуће испрограмирати и “пешке” тако што ћемо дефинисати своју функцију претраге. Наравно, у реалној ситуацији је увек боље употребити библиотечке функције јер се код једноставније пише, мања је могућност грешке, а алгоритми на којима су засноване библиотечке функције су често напреднији од наивних алгоритама на којима се “пешке” решења понекад заснивају, какво је и ово које ћемо описати у наставку.

Размотримо, на пример, како бисмо у речи `ababcbab` тражили реч `abc`.

- Прво можемо проверити да ли се реч `abc` налази у речи `ababcbab` почевши од позиције 0. То радимо тако што поредимо редимо слова речи `abc` и слова речи `ababcbab` почевши од позиције 0, све док не пронађемо различито слово или док не дођемо до краја речи `abc` а не пронађемо разлику. У овом примеру разлика ће бити пронађена на позицији 2 (слова `a` на позицији 0 и `b` на позицији 1 ће бити једнака, а онда ће се слово `c` разликовати од слова `a`). Пошто смо нашли разлику пре него што смо стигли до краја речи `abc`.
- Настављамо претрагу тако што проверавамо да ли се можда реч `abc` налази у речи `ababcbab` почевши од позиције 1. Овај пут поредимо слово на позицији 0 у речи `abc` са словом на позицији 1 у речи `ababcbab` и одмах наилазимо на разлику, јер `a` није једнако `b`.
- Након тога претрагу настављамо тако што проверавамо да ли се реч `abc` налази у речи `ababcbab` почевши од позиције 2. Тада поредимо слово `a` на позицији 0 у речи `abc`, са словом `a` на позицији 2 у речи `ababcbab`, слово `b` на позицији 1 у речи `abc`, са словом `b` на позицији 3 у речи `ababcbab` и слово `c` на позицији 2 у речи `abc`, са словом `c` на позицији 4 у речи `ababcbab` и пошто не наилазимо на разлику, констатујемо да је подреч пронађена.

Ако бисмо, на пример, тражили реч `cba` у речи `ababcbab` тада бисмо по истом принципу кренули да је тражимо од позиције 0, затим 1 итд. Када реч `cba` не пронађемо ни тако да почиње од позиције 4 речи `ababcbab` (јер `cab` није исто што и `cba`) нема потребе да тражимо даље, јер иза позиције 4 нема довољно карактера да би се реч `abc` пронашла.

Дакле, за сваку позицију i у речи `plast`, од позиције 0, па док важи да је збир позиције i и дужине речи `igla` мањи или једнак дужини речи `plast` проверавамо да ли се реч `igla` налази у речи `plast` почевши од позиције i . То радимо тако што спроводимо алгоритам линеарне претраге који редом, за сваку позицију j у речи `igla` проверава да ли је слово у речи `igla` на тој позицији различито од слова на позицији $i + j$ у речи `plast`. Алгоритам претраге имплементирамо на неки од начина који смо приказали у задатку **Негативан број**. На пример, у петљи увећавамо j од нуле, па све док је j мање од дужине речи `igla` и док важи да су слова у речи `igla` на позицији j и речи `plast` на позицији $i + j$ једнака. Када се петља заврши, проверавамо зашто је прекинута. Ако утврдимо да је вредност j једнака дужини речи `igla`, можемо констатовати да разлика није пронађена, да се прво појављивање речи `igla` у речи `plast` налази на позицији i и функција може да врати резултат i (прекидајући тиме спољашњу петљу). У супротном је пронађено различито слово, и наставља се са извршавањем спољашње петље (i се увећава, и подниска се тражи од следеће позиције, ако евентуално није достигнут услов за завршетак спољашње петље).

Још један начин да имплементирамо претрагу је да уведемо логичку променљиву која бележи да ли је разлика пронађена и коју иницијализујемо на `false`. У сваком кораку унутрашње петље (петље по j) проверавамо да ли је су слова у речи `igla` на позицији j и речи `plast` на позицији $i + j$ једнака. Ако нису, логичкој променљивој постављамо вредност `true` и прекидамо унутрашњу петљу. Након унутрашње петље проверавамо да ли је вредност логичке променљиве остала `false` и ако јесте, функција може да врати вредност i .

```
// Trazimo "iglu u plastu" tj. trazimo poziciju u reci plast
// na kojoj se nalazi rec igla.
// Ako takva pozicija ne postoji, funkcija vraca -1.
int podniska(string igla, string plast) {
```

```

// za svaku poziciju u reci plast takvu da do kraja
// ima bar onoliko karaktera koliko ima u reci igla
for (size_t i = 0; i + igla.size() <= plast.size(); i++) {
    // proveravamo da li se rec igla nalazi u reci plast
    // pocevsi od pozicije i
    bool razliciti = false;
    for (size_t j = 0; j < igla.size(); j++) {
        // proveravamo da li je j-to slovo reci igla razlicito od j-tog
        // slova reci plast krenuvsi od pozicije i
        if (plast[i+j] != igla[j]) {
            razliciti = true;
            break;
        }
    }
    // ako smo dosli do kraja reci igla, tada smo je pronasli
    if (!razliciti)
        return i;
}
// nismo nasli rec igla i vracamo -1
return -1;
}

int main() {
    string plast, igla;
    cin >> plast >> igla;
    cout << podniska(igla, plast) << endl;
    return 0;
}

```

Рецимо да је описан алгоритам било могуће спровести и над нискама које су терминисане нулом, једино што се у петљи не би ишло до дужине ниске (она није унапред позната и потребно је проћи кроз целу ниску да би се пронашла нула и да би јој се израчунала дужина), већ би се итерација вршила док карактер на текућој позицији не постане нула. Итерација кроз ниску *s* терминисану нулом, канонски се врши на следећи начин:

```

for (int i = 0; s[i] != '\0'; i++)
    ...

```

или још једноставније

```

for (int i = 0; s[i]; i++)
    ...

```

На крају, рецимо и да се овај проблем може решавати и много ефикаснијим специјализованим алгоритмима какви су Кнут-Морис-Пратов алгоритам или Бојер-Муров алгоритам.

```

// trazimo "iglu u plastu" tj. trazimo poziciju u reci plast na kojoj
// se nalazi rec igla. Ako takva pozicija ne postoji, funkcija vraca -1
int podniska(char igla[], char plast[]) {
    // za svaku poziciju u reci plast
    for (int i = 0; plast[i] != '\0'; i++) {
        // Proveravamo da li se rec igla nalazi u reci plast pocevsi od
        // pozicije i. Brojac j uvecavamo dok ne dodjemo do kraja neke
        // reci ili do prve razlike.
        int j;
        for (j = 0; igla[j] != '\0' && plast[i+j] != '\0' &&
            plast[i+j] == igla[j]; j++)
            ;
        // ako smo dosli do kraja reci igla, tada smo nasli njeno
        // pojavljivanje
        if (igla[j] == '\0')
            return i;
    }
}

```

```

    }
    // nismo nasli rec igla i vracamo -1
    return -1;
}

```

Решење грубом силом може бити засновано и на на библиотечкој функцији која проверава једнакост два распона елемената низа. У језику C++ то је функција `equal`.

```

// Trazimo "iglu u plastu" tj. trazimo poziciju u reci plast na kojoj
// se nalazi rec igla. Ako takva pozicija ne postoji, funkcija vraca
// -1.
int podniska(string igla, string plast) {
    // za svaku poziciju u reci plast takvu da do kraja ima bar onoliko
    // karaktera koliko ima u reci igla
    for (size_t i = 0; i + igla.size() <= plast.size(); i++) {
        // proveravamo da li se rec igla nalazi u reci plast pocevsi od
        // pozicije i
        if (equal(igla.begin(), igla.end(),
                  next(plast.begin(), i)))
            return i;
    }
    // nismo nasli rec igla i vracamo -1
    return -1;
}

```

Задатак: Подниске

Напиши програм који одређује колико пута се дата ниска јавља као подниска друге дате ниске. Више појављивања подниске се могу преклапати.

Улаз: Са стандардног улаза се уносе две ниске, свака у посебном реду. Прва ниска има највише 10 карактера, а друга највише 1000.

Излаз: На стандардни излаз исписати тражени број појављивања.

Пример 1

Улаз	Излаз
aba	4
ababababababac	

Пример 2

Улаз	Излаз
aa	3
aaaa	

Решење

Претрага од сваке позиције

Директан алгоритам да се овај задатак реши је да се покуша претрага подниске од сваке позиције кренувши од нулте, па до оне позиције на којој се десни крај ниске поклапа са десним крајем подниске. Обилазак позиција, дакле, тече све док је збир текуће позиције и дужине подниске мањи или једнак од дужине ниске. За сваку такву позицију проверавамо да ли на њој почиње подниска и ако почиње, увећавамо бројач иницијално постављен на нулу.

Проверу да ли се подниска налази у датој ниски кренувши од дате позиције можемо урадити на разне начине. Један начин је да издвојимо део ниске од дате позиције помоћу функције `substr` и да га упоредимо са датом подниском помоћу оператора `==`.

Овај приступ може бити неефикасан јер се непотребно сваки пут гради нова ниска пре него што се изврши поређење.

У језику C++ можемо употребити функцију `equal` која проверава једнакост два опсега исте дужине.

За поређење ниски (тј. њихових подниски) можемо и сами испрограмирати функцију која коришћењем линеарне претраге (попут оне у задатку **Погођена комбинација**) проверава да ли постоји карактер у којима се две ниске разликују, при чему функција уз ниске `s` и `t` које се пореде, може добити и индексе позиција `ps` и `pt` од којих ће се поређење вршити и дужину `n` делова ниски који се пореде. У петљи чији бројач `i` креће од 0 и тече све док је мањи од `n` пореде се карактери на позицијама `s[ps + i]` и `t[pt + i]` и ако су различити функција враћа `false`. Ако се петље заврше без пронађене разлике делови су једнаки и функција враћа `true`.

Уместо засебне функције такву претрагу можемо реализовати и у главној функцији у телу петље која набраја потенцијалне позиције почетка, али нам је тада за имплементацију линеарне претраге потребна помоћна логичка променљива и неки начин да унутрашњу петљу прекинемо када се наиђе на први различити карактер.

```
string niska, podniska;
cin >> podniska >> niska;
int brojPojavljivanja = 0;
for (int p = 0; p + podniska.length() <= niska.length(); p++)
    if (equal(podniska.begin(), podniska.end(), next(niska.begin(), p)))
        brojPojavljivanja++;
cout << brojPojavljivanja << endl;
```

Библиотека претрага подниске

Библиотека функција за претрагу подниске обично има додатни параметар којим се контролише позиција од које претрага треба да крене. То важи за методу `find` у језику C++, која враћа позицију првог пронађеног појављивања подниске (које није испред наведене позиције) или специјалну ознаку `string::npos` када нема појављивања подниске који нису испред наведене позиције. Сваки пут када се пронађе подниска, увећава се бројач, а нова претрага креће од позиције непосредно иза позиције где је подниска пронађена.

Рецимо да би се пажљивом анализом ниске која се тражи то могло оптимизовати. На пример, ако смо нашли ниску `abcabcabc` унутар шире ниске, ново појављивање те подниске не треба тражити од наредне позиције (позиције првог карактера `b`), већ од треће позиције на којој се појављује карактер `a` (то је најдужи суфикс који је уједно и префикс ниске која се тражи). Ово је основа алгорита КМП о ком ће бити више речи у наредном тому ове збирке.

```
string niska, podniska;
cin >> podniska >> niska;
int brojPojavljivanja = 0;
int p = 0;
while ((p = niska.find(podniska, p)) != string::npos) {
    brojPojavljivanja++;
    p++;
}
cout << brojPojavljivanja << endl;
```

Задатак: Немењајуће ротације

Ниска дужине n се циклично помера за једно место у лево тако што се њен почетни карактер са почетка пребаци на крај. Након n цикличних померања ниска се враћа у почетни положај. На пример, сва циклична померања ниске `abcabc` су `bcabca`, `cabcab`, `abcabc`, `bcabca`, `cabcab` и `abcabc`. Од њих су два циклична померања једнака полазној ниски.

Улаз: Са стандардног улаза се учитава ниска састављена само од малих слова енглеске абетецеде.

Излаз: На стандардни излаз исписати колико је њених цикличних померања једнако почетној ниски.

Пример 1

Улаз	Излаз
algoritmi	0

Пример 2

Улаз	Излаз
dadadada	4

Решење

Задатак можемо решити грубом силом, тако што у сваком кораку ниску ротирамо за једном место улево и проверавамо да ли је једнака полазној.

Сложеност таквог решења је $O(n^2)$, где је n дужина ниске s .

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

```
using namespace std;
```

```
int main() {
```



```

string s;
cin >> s;
int n = s.size();
int broj = 0;
string ss = s;
for (int i = 0; i < n; i++) {
    if (ss == s)
        broj++;
    rotate(begin(ss), next(begin(ss)), end(ss));
}
cout << broj << endl;
return 0;
}

```

Једно ефикасно решење се заснива на томе да ће се свако циклично померање ниске s појавити као подниска ниске ss (почетно померање ће се појавити два пута). Овим се проблем своди на пребројавање појављивања ниске s унутар ниске ss , што се ефикасно може урадити КМП алгоритмом.

Сложеност овог приступа је $O(n)$, где је n дужина ниске s .

```

#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main() {
    string s;
    cin >> s;
    vector<int> kmp(s.size() + 1, 0);
    kmp[0] = -1;
    for (size_t i = 0; i < s.size(); i++) {
        int k = i;
        while (k > 0 && s[i] != s[kmp[k]])
            k = kmp[k];
        kmp[i + 1] = kmp[k] + 1;
    }

    int broj = 0;
    string ss = s + s;
    size_t i = 0, j = 0;
    while (i < ss.size()) {
        if (ss[i] == s[j]) {
            i++; j++;
            if (j == s.size()) {
                broj++;
                j = kmp[j];
            }
        } else if (j == 0)
            i++;
        else
            j = kmp[j];
    }

    cout << broj - 1 << endl;
    return 0;
}

```

Задатак: Периодичност ниске

Реч w је периодична ако постоји непразна реч $p = p_1p_2$ и природан број $n \geq 2$ тако да је $w = p^n p_1$. На пример, реч *abacabacabacab* је периодична јер се понавља *abac*, при чему се последње понављање не завршава цело већ се зауставља са *ab*, тј. реч је $(abac)^3 ab$. Напиши програм који проверава да ли је унета реч периодична.

Улаз: Прва линија стандардног улаза садржи реч која се састоји само од малих слова енглеског алфабета – њих највише 50000.

Излаз: На стандардни излаз исписати реч *da* ако реч јесте периодична тј. *ne* ако није.

Пример

<i>Улаз</i>	<i>Излаз</i>
abbaabbaabbaa	da

Решење**Груба сила**

Задатак можемо решити грубом силом, тако што ћемо за сваку вредност d такву да је $2d \leq |w|$ проверити да ли је реч периодична при чему је период префикс речи w дужине d . Једноставно се доказује да је реч периодична са периодом p чија је дужина d , ако и само ако за свако i за које је $0 \leq i$ и $i + d < |w|$ важи да је $w_i = w_{i+d}$. Задатак се онда решава са две угнежђене линеарне претраге – у спољној проверавамо све потенцијалне вредности дужине d , а у унутрашњој проверавамо да ли постоји вредност i таква да је $w_i \neq w_{i+d}$. Ако у унутрашњој петљи утврдимо да такво i не постоји, тада је ниска периодична. Ако пронађемо такво i , можемо прекинути унутрашњу петљу (реч није периодична са периодом дужине d) и прећи на следеће d (за један веће). Ако такво d не постоји, тада можемо констатовати да реч није периодична.

Сложеност најгорег случаја овог алгоритма је квадратна. Заиста, унутрашња линеарна претрага може у најгорем случају захтевати $O(|w|)$ итерација, и она се понавља $O(|w|)$ пута. Ипак, ако је ниска насумична, реално је очекивати да ће се за већину вредности d веома брзо установљавати да је $w_i \neq w_{i+d}$, па програм може радити доста брже од најгорег случаја.

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

using namespace std;

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

    string str;
    cin >> str;

    bool periodicna = false;
    for (int p = 1; 2 * p <= str.size(); p++) {
        bool greska = false;
        for (int i = 0; i + p < str.size(); i++)
            if (str[i] != str[i + p]) {
                greska = true;
                break;
            }
        if (!greska) {
            periodicna = true;
            break;
        }
    }

    cout << (periodicna ? "da" : "ne") << endl;

    return 0;
}
```

Кнут-Морис-Пратов алгоритам

Ефикасно решење се заснива на теореми која каже да је ниска w периодична ако и само ако има прави префикс-суфикс чија је дужина најмање половина ниске, тј. ако постоје непразни x , s и t такви да је $xs = tx$ и $2 \cdot |x| \geq |w|$. На пример, ако је ниска `абасабасаба`, тада је тражени префикс-суфикс x једнак, `абасаба` остатак s једнак је `саба`, док је t једнак `абас`.

Докажимо претходну карактеризацију. Прво, претпоставимо да је ниска периодична. Тада постоји непразно $p = p_1p_2$ тако да је $x = p^n p_1$, за неко $n \geq 2$. Тада је $t = p_1p_2 = p$, $x = p^{n-1}p_1$, док је $s = p_2p_1$. Важи да је $|x| = (n-1)|p| + |p_1|$, а пошто је $n \geq 2$, важи да је $(n-1) \cdot |p| \geq |p|$, па је $|x| \geq |t|$ и $2 \cdot |x| \geq |x| + |t| = |w|$.

Докажимо и супротан смер.

Докажимо прво индукцијом да ако важи да је $w = xs = tx$ за непразне t и s , тада постоје речи u и v тако да је $t = uv$, $s = vu$ и $w = (uv)^n$. Претпоставимо да тврђење важи за све речи дужине мање од k и претпоставимо да је реч w дужине k . Ако је $|x| < |t|$, тада постоји y тако да је $t = xy$, па на основу $xs = tx$ следи и да је $s = yx$. Тражене речи u и v су тада x и y . Ако је $|x| \geq |t|$ тада постоји y тако да је $x = ty$, па на основу $xs = tx$ следи $ys = x$ тј. $ys = ty$. Пошто је s и t непразно важи да је $|ty| = |x| < |w|$, па на основу индуктивне хипотезе постоје u и v такви да је $t = uv$, $s = vu$ и n такво да је $y = (uv)^n u$. Тада је $x = ty = uv(uv)^n u = (uv)^{n+1}u$, па тврђење следи.

Докажимо и да је реч периодична. Нека су u , v и n такви да је $t = uv$, $s = vu$ и $w = (uv)^n u$ (они постоје на основу претходне дискусије). Прво, $n \geq 1$ (ако би важило $n = 0$, тада би важило да је $w = s = t = u$, а да је x празна реч, па одатле следи да је $|t| \geq 1 > |x| = 0$, што је контрадикција са претпоставком да је $|x| \geq |t|$). Ако би важило да је $n = 1$, тада би важило да је $w = uvu$, па из тј. $w = tx = xs$ мора да важи да је $x = u$, што са $|x| \geq |t|$ повлачи да је $t = s = x = u$. Тада је $p = p_2 = u$, док је p_1 празно, па је $w = u^2$ и периодична је. Преостаје још случај $n \geq 2$. Међутим, тада се може узети да је $p_1 = u$, $p_2 = v$, $p = uv$ и важи да је $w = p^n u$, при чему је $n \geq 2$, па је реч поново периодична.

Дакле, решење се заснива на томе да пронађемо дужину d најдужег правог суфикса речи w који је уједно њен префикс, и да се провери да ли важи да је $2d \geq |w|$. То можемо урадити помоћу Кнут-Морис-Пратовог алгоритма (исто као у задатку [Префикс суфикс](#)).

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

using namespace std;

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

    string str;
    cin >> str;

    vector<int> kmp(str.size() + 1, 0);
    kmp[0] = -1;
    for (int i = 0; i < str.size(); i++) {
        int k = i;
        while (k > 0 && str[i] != str[kmp[k]])
            k = kmp[k];
        kmp[i + 1] = kmp[k] + 1;
    }

    if (2 * kmp[str.size()] >= str.size())
        cout << "da" << endl;
    else
        cout << "ne" << endl;

    return 0;
}
```

3.2 Најдужи палиндроми

Задатак: Најдужа палиндромска подниска

Дат је стринг s који садржи само мала слова. Приказати најдужи сегмент стринга s (низ узастопних елемената стринга) који је палиндром у стрингу s . Ако има више најдужих сегмената приказати сегмент чији почетак има најмањи индекс.

Улаз: Прва и једина линија стандардног улаза садржи стринг састављен од малих слова.

Израз: На стандардном излазу приказати први најдужи сегмент датог стринга који је палиндром.

Пример 1

Улаз Израз
cabbadcmmc abba

Пример 2

Улаз Израз
babcbabcbaccba abcbabcb

Решење

Провера свих сегмената

Задатак је могуће решити анализом свих сегмената, провером да ли је текући сегмент палиндром и одређивањем најдужега пронађеног палиндрома.

Пошто је провера палиндрома сложености $O(n)$, а сегмената има $O(n^2)$, сложеност овог приступа је $O(n^3)$.

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

using namespace std;

// provera da li je s[i, j] palindrom
bool palindrom(const string& s, int i, int j){
    while (i < j && s[i] == s[j]) {
        i++; j--;
    }
    return i >= j;
}

int main() {
    string s;
    cin >> s;
    int n = s.size();

    int maxDuzina = 0, maxPocetak = 0;
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++)
            if (palindrom(s, i, j)) {
                int duzina = j - i + 1;
                if (duzina > maxDuzina) {
                    maxDuzina = duzina;
                    maxPocetak = i;
                }
            }

    cout << s.substr(maxPocetak, maxDuzina) << endl;

    return 0;
}
```

Провера свих сегмената у редоследу опадајуће дужине

Имплементација се може мало поједноставити и дугачки палиндроми се могу пронаћи брже, ако се примети да сегменте можемо анализирати редом почев од најдужег сегмента па уназад до сегмента дужине 1 (слично као у задатку **Све подречи по опадајућој дужини**). Приметимо да ће са овим редоследом обиласка први сегмент који је палиндром управо бити најдужи палиндром који тражимо. Приметимо да излаз из угнежђених петљи није могуће остварити наредбом `break` (тима би се изашло из унутрашње, али не и спољашње петље), већ излаз морамо реализовати помоћу помоћне логичке променљиве (слично као у случају алгоритама претраге, као, на пример, у задатку **Негативан број**) или наредбом `goto` (иако је потребно избегавати је, неки аутори сматрају да је прекид угнежђених петљи једина ситуација у којој је употреба `goto` оправдана). Сложеност најгорег случаја овог приступа је и даље $O(n^3)$.

```
#include <iostream>

using namespace std;

// provera da li je s[i, j] palindrom
bool palindrom(const string& s, int i, int j){
    while (i < j && s[i] == s[j]) {
        i++; j--;
    }
    return i >= j;
}

int main() {
    string s;
    cin >> s;
    // duzina niske s
    int n = s.size();
    // potrebno za prekid dvostruke petlje
    bool nasli = false;
    // proveravamo sve duzine redom
    for(int d = n; d >= 1 && !nasli; d--) {
        // proveravamo rec odredjenu indeksima [p, p + d - 1]
        for(int p = 0; p + d - 1 < n && !nasli; p++) {
            // ako smo naisli na palindrom
            if (palindrom(s, p, p + d - 1)) {
                // ispisujemo ga
                cout << s.substr(p, d) << endl;
                // prekidamo dvostruku petlju
                nasli = true;
            }
        }
    }

    return 0;
}
```

Провера центара

Палиндроми поседују одређено својство инкременталности које нам може помоћи да пронађемо ефикаснији алгоритам. Наиме, ако је познат центар палиндрома (то може бити било неко слово, било позиција тачно између два суседна слова) и ако знамо да се k слова око тог центра сликају као у огледалу (и тиме граде палиндром), онда за проверу да ли се $k + 1$ слова око тог центра сликају као у огледалу не треба проверавати све из почетка, већ је довољно само проверити да ли су два слова на спољним позицијама ($k + 1$. слово лево тј. десно од центра) једнака. Зато ефикасније решење добијамо ако за свако слово свако слово речи одредимо најдужи палиндром непарне дужине такав да му је изабрано слово центар и за сваку позицију између два слова одредимо најдужи полином парне дужине којима је та позиција центар.

Да бисмо одредили палиндром са центром у слову s_i , ширимо палиндром s_i у десно и у лево за k слова док

3.2. НАЈДУЖИ ПАЛИНДРОМИ

се налазимо унутар речи ($i - k \geq 0, i + k < n$) и док су одговарајућа слова једнака ($s_{i-k} = s_{i+k}$). У тренутку када се то први пут наруши добијамо најдужи палиндром са центром у s_i (ако се изађе из речи даље проширивање није могуће, а ако се пронађе различит пар слова даља проширивања не могу више да дају палиндром). Одређивање најдужег палиндрома са центром између два слова вршимо на веома сличан начин.

За сваку позицију (а њих има $2n - 1$ тј. $O(n)$) налазимо најдужи палиндром са центром на њој ширећи текући палиндром налево и надесно и глобално најдужи палиндром налазимо као најдужи од тих палиндрома.

Ширење се обавља једним проласком и захтева време $O(n)$ и укупна сложеност алгоритма је $O(n^2)$.

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

int main() {
    string s;
    cin >> s;
    // duzina učitane reci
    int n = s.size();

    // duzina i pocetak najduzeg palindroma
    int maxDuzina = 0, maxPocetak = 0;

    // prolazimo kroz sva slova reci
    for (int i = 0; i < n; i++) {
        int duzina, pocetak;

        // nalazenje najduzeg palindroma neparne duzine ciji je centar
        // slovo s[i]
        int k = 1;
        while (i - k >= 0 && i + k < n && s[i - k] == s[i + k])
            k++;
        // duzina i pocetak maksimalnog palindroma
        duzina = 2 * k - 1;
        pocetak = i - k + 1;

        // azuriramo maksimum ako je to potrebno
        if (duzina > maxDuzina) {
            maxDuzina = duzina;
            maxPocetak = pocetak;
        }

        // nalazenje najduzeg palindroma parne duzine ciji je centar
        // izmedju slova s[i] i s[i+1]
        k = 0;
        while (i - k >= 0 && i + k + 1 < n && s[i - k] == s[i + k + 1])
            k++;
        // duzina i pocetak maksimalnog palindroma
        duzina = 2 * k;
        pocetak = i - k + 1;

        // azuriramo maksimum ako je to potrebno
        if (duzina > maxDuzina) {
            maxDuzina = duzina;
            maxPocetak = pocetak;
        }
    }

    // izdvajamo i ispisujemo odgovarajuci palindrom
    cout << s.substr(maxPocetak, maxDuzina) << endl;
```

```
return 0;
}
```

Допуна речи и позиције

Постоји неколико техника које могу да мало скрате имплементацију претходног алгоритма, обједињавајући случајеве палиндрома парне и непарне дужине. Замислимо да се пре првог, након последњег и између свака два суседна слова речи постави специјални карактер `|`. На пример, реч `aabcbab` допуњавамо до речи `|a|a|b|c|b|a|b|`. На тај начин добијамо то да су сада сви центри палиндрома карактери овако допуњеног речи и довољно је анализирати само палиндроме непарне дужине у њему. Ово допуњавање је могуће реализовати и физички (у програму креирати допуњени стринг), што може мало да олакша имплементацију по цену мало споријег програма (додуше не асимптотски) и додатног заузећа меморије. Ипак наредно разматрање нам говори да ове помоћне карактере можемо само разматрати док разматрамо алгоритам, без експлицитног прављења допуњеног стринга у програму.

Да бисмо олакшали излагање индексе у допуњеној речи ћемо називати позиције, а у оригиналној речи само индекси. На пример,

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	-	pozicije
	a		a		b		c		b		a		b			
	0		1		2		3		4		5		6		-	indeksi

Приметимо неколико чињеница. Ако је полазна реч дужине n , укупно имамо $N = 2n + 1$ позицију. Слова полазне речи се налазе на непарним позицијама, док се на парним позицијама налази специјални знак `|`. Слово са индексом k се налази на позицији $p = 2k + 1$, што значи да се на непарној позицији p налази слово оригиналне речи са индексом $k = \lfloor \frac{p}{2} \rfloor$.

За сваку позицију i ($0 \leq i < N$) дужину палиндрома са центром на тој позицији можемо одредити на исти начин, без обзира на то да ли је на тој позицији слово или специјални знак `|`. Рецимо да ћемо овде подразумевати дужину палиндрома у оригиналној речи (а не допуњеној) и она је једнака броју карактера са леве стране дате позиције у допуњеној речи који су једнаки одговарајућим карактерима са десне стране те позиције у допуњеној речи. На пример, у претходном примеру за позицију 7 то је 3, јер је палиндром `bcb` дужине 3, што одговара томе да три карактера `|b|` са леве стране карактера `c` у допуњеној речи одговарају карактерима `|b|` са десне стране карактера `c`.

На почетку, дужину палиндрома d постављамо на 1, ако је позиција непарна тј. 0 ако је парна. Заиста, ако је позиција непарна, на њој се налази слово које је само за себе палиндром дужине 1 (из другог угла гледано, лево и десно од ње се налазе `|`, па је бар један карактер једнак). Ако је позиција парна, око ње се налазе два слова (осим у случају крајњих позиција) и не знамо унапред да ли су она једнака, тако да дужину палиндрома иницијално постављамо на 0. Овим постижемо да су бројеви $i + d$ и $i - d$ парни, што значи да су $i - d - 1$ и $i + d + 1$ непарни и ако су у опсегу $[0, N)$, они указују на наредна два карактера чију једнакост треба проверити. Ако су карактери на одговарајућим индексима једнаки (то су индекси $\lfloor \frac{i-d-1}{2} \rfloor$ и $\lfloor \frac{i+d+1}{2} \rfloor$) онда се d увећава за 2 (из једног угла гледано, та два једнака карактера се додају текућем палиндрому па му се дужина повећава за 2, а из другог угла гледано, испред првог и иза другог се налазе специјални знаци `|` који су сигурно једнаки и њихову једнакост није потребно експлицитно проверавати). Тиме се одржава и инваријанта да су бројеви $i + d$ и $i - d$ парни и петља се може наставити на исти начин све док се наиђе на два различита слова или се не изађе ван опсега допуњене речи.

Рецимо још и да се и провера припадности индекса тј. позиција опсегу речи може елиминисати ако се полазна реч прошири са додатним специјалним карактерима на почетку и на крају (они морају бити различити и обично се користе `^` и `$`, јер се ти карактери користе за означавање почетка и краја у регуларним изразима).

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

```
using namespace std;
```

```
string dopuni(const string& s) {
    string rez = "";
    for (int i = 0; i < s.size(); i++)
        rez += "|" + s.substr(i, 1);
    rez += "|$";
}
```

3.2. НАЈДУЖИ ПАЛИНДРОМИ

```
return rez;
}

int main() {
    string s;
    cin >> s;
    string t = dopuni(s);

    // dovoljno je pronaci najveći palindrom neparne dužine u proširenoj reci

    int maxDuzina = 0, maxCentar = 0;
    // proveravamo sve pozicije u dopunjenoj reci
    for (int i = 1; i < t.size() - 1; i++) {
        // prosirujemo palindrom sa centrom na poziciji i dokle god je to
        // moguće
        int d = 0;
        while (t[i - d - 1] == t[i + d + 1])
            d++;

        // azuriramo maksimum ako je potrebno
        if (d > maxDuzina) {
            maxDuzina = d;
            maxCentar = i;
        }
    }

    // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
    int maxPocetak = (maxCentar - maxDuzina) / 2;
    cout << s.substr(maxPocetak, maxDuzina) << endl;
}
```

Маначеров алгоритам

Посматрајмо сада како изгледа дужина најдужега палиндрома за сваку од позиција у речи babcbabcbaccba. Обележимо ову дужину са d_p . Обележимо допуњену реч са t .

0	1	2	3	4	5	6	7	8	9	10	11	12	1
b	a	b	c	b	a	b	c	b	a	c	c	b	a
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 25	26 27
0 1	0 3	0 1	0 7	0 1	0 9	0 1	0 5	0 1	0 1	0 1	2 1	0 1	0 1

Посматрајмо палиндром са центром на позицији 11 - његова дужина је $d_{11} = 9$ и простире се од позиције 2 до позиције 20.

Посматрајмо сада дужину најдужега палиндрома са центром на позицији 12. Пошто је наш палиндром симетричан око позиције 11, позицији 12, одговара позиција 10. Знамо да је $d_{10} = 0$. То је зато што је $t_9 \neq t_{11}$. Међутим, ми знамо да је $t_9 = t_{13}$ (пошто су обе унутар нашег палиндрома), па је зато $t_{11} \neq t_{13}$ и важи да је $d_{12} = d_{10} = 0$. Приметимо да ово можемо констатовати без икакве потребе за упоређивањем карактера.

Посматрајмо сада дужину најдужега палиндрома са центром на позицији 13. Њему одговара палиндром са центром на позицији 9. Важи да је $d_9 = 1$, јер је $t_8 = t_{10}$ и $t_7 \neq t_{11}$. На основу симетрије палиндрома са центром у 11, важи да је $t_8 = t_{14}$, да је $t_{10} = t_{12}$, $t_7 = t_{15}$ и да је $t_{11} = t_{11}$. Зато је $t_{14} = t_{12}$ и $t_{15} \neq t_{11}$, па је $d_{13} = d_9 = 1$.

Наизглед, важи да је за свако i унутар ширег палиндрома са центром у некој позицији C број d_i једнак броју $d_{i'}$, где се i' одређује као симетрична позиција позицији i у односу на C (важи да је растојање од C до i и i' једнако, па је тј $C - i' = i - C$, тј. $i' = C - (i - C)$). Но, то није увек тачно.

Посматрајмо d_{15} и њему одговарајућу вредност d_7 . Оне нису једнаке. Зашто? Посматрајмо шта је оно то можемо закључити из симетрије палиндрома са центром на позицији 11. Важи да је t_2 до t_{12} једнако одговарајућим карактерима t_{20} до t_{10} - то је гарантовано симетријом и није потребно проверавати. Међутим,

важи да је $d_7 = 7$. Знамо зато и да је $t_1 = t_{13}$, међутим, не можемо да тврдимо да је $t_{21} = t_9$, зато то t_{21} није више део палиндрома са центром на позицији C - проверу да ли је $t_{21} = t_9$ је потребно посебно извршити.

Дакле, важи следеће. Претпоставимо да је $[L, R]$ палиндром са центром на позицији C (тада је $C - L = R - C$), да је i неки индекс унутар тог палиндрома (нека је $C < i < R$) и да је $i' = C - (i - C)$ њему симетричан индекс. Ако је палиндром са центром у i' садржан у палиндрому (L, R) (без урачунатих крајева) тј. ако је $L < i' - d_{i'}$, тј. $d_{i'} < i' - L = (C - (i - C)) - (C - (R - C)) = R - i$ тј. $d_{i'} < R - i$, тада је $d_i = d_{i'}$. Докажимо ово. За сваку вредност $0 \leq j \leq d_{i'}$ треба доказати да је $t_{i-j} = t_{i+j}$. Заиста, пошто важи да је $L < i' - j$ и да је $i + j < R$, важи да је $t_{i-j} = t_{i'+j}$ и да је $t_{i+j} = t_{i'-j}$. Међутим, пошто је i' центар палиндрома дужине $d_{i'}$, важи да је $t_{i'-j} = t_{i'+j}$. Зато је на позицији i центар палиндрома дужине бар $d_{i'}$. Докажимо да је ово и горње ограничење, тј. докажимо да је $t_{i-d_{i'}-1} \neq t_{i+d_{i'}+1}$. Пошто је $i + d_{i'} < R$, важи да је $i + d_{i'} + 1 \leq R$, па је $t_{i-d_{i'}-1} = t_{i'+d_{i'}+1}$ и $t_{i+d_{i'}+1} = t_{i'-d_{i'}-1}$. Међутим, пошто је палиндром са центром у i' дужине $d_{i'}$ важи да је $t_{i'-d_{i'}-1} \neq t_{i'+d_{i'}+1}$.

Ако је $[L, R]$ палиндром са центром на позицији C и ако је i неки индекс унутар тог палиндрома ($C < i < R$), али такав да је $d_{i'} \geq R - i$, онда можемо само да закључимо да је $d_i \geq R - i$.

Ово инспирише наредни алгоритам, познат под именом Маначеров алгоритам. Слично као у претходној верзији обрађујемо све позиције i од 0 до $N - 1$. При том одржавамо индексе C и R такве да је $[C - (R - C), R]$ палиндром. Ако је $i \geq R$, тада палиндром са центром у i одређујемо из почетка, повећавајући за два дужину палиндрома d_i која креће од 0 или 1 (у зависности од парности позиције i), све док је то могуће, исто као у претходно описаном алгоритму. Међутим, ако је $i < R$, тада одређујемо $i' = C - (i - C)$ и ако важи да је $d_{i'} < R - i$, постављамо одмах $d_i = d_{i'}$. Ако је $d_{i'} \geq R - i$, тада дужину d_i постављамо на почетну вредност $R - i$ тј. на $R - i + 1$ тако да су $i - d_i$ и $i + d_i$ парни бројеви, и онда је постепено повећавамо за 2, све док је то могуће (опет, веома слично као у претходно описаном алгоритму). Ако је пронађени палиндром са центром у i такав да му десни крај превазиђе позицију R , онда њега проглашавамо за нови палиндром $[L, R]$, постављајући му центар $C = i$ и десни крај $R = i + d_i$. На почетку можемо иницијализовати $R = C = 0$ (тима обезбеђујемо да не може да важи да је $i < R$ и да се на почетку неће користити симетричност окружујућег полинома).

Могуће је доказати да је сложеност овог алгоритма $O(n)$ (интуитивно, проналажење кратких палиндрома захтева мали број извршавања унутрашње петље, док је проналажење једног дугачког палиндрома захтева дуже извршавање унутрашње петље, али доводи до тога да ће се у наредним корацима у великом броју случајева у потпуности избегава њено извршавање).

```
#include <iostream>
```

```
#include <string>
```

```
#include <vector>
```

```
using namespace std;
```

```
// da bismo uniformno posmatrali palindrome i parne i neparne duzine,
```

```
// prosirujemo string dodajuci ^ i $ oko njega i umecuci | izmedju
```

```
// svih slova npr. abc -> ^a|b|c$
```

```
string dopuni(const string& s) {
    string rez = "^";
    for (int i = 0; i < s.size(); i++)
        rez += "|" + s.substr(i, 1);
    rez += "$";
    return rez;
}
```

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

```
// jednostavnosti radi dopunjavamo rec s
```

```
string t = dopuni(s);
```

```
// d[i] je najveći broj takav da je [i - d[i], i + d[i]] palindrom
```

```
// to je ujedno i dužina najdužeg palindroma čiji je centar na
```

```
// poziciji i (pozicije su ili slova originalnih reci, ili su
```

```
// izmedju njih)
vector<int> d(t.size());
// znamo da je [L, R] palindrom sa centrom na poziciji C
int C = 0, R = 0; // L = C - (R - C)
for (int i = 1; i < t.size() - 1; i++) {
    // karakter simetrican karakteru i u odnosu na centar C
    int i_sim = C - (i - C);
    if (i < R && i + d[i_sim] < R)
        // nalazimo se unutar palindroma [L, R], ciji je centar C
        // palindrom sa centrom u i_sim i palindrom sa centrom u i su
        // celokupno smesteni u palindrom (L, R)
        d[i] = d[i_sim];
    else {
        // ili se ne nalazimo u okviru nekog prethodnog palindroma ili
        // se nalazimo unutar palindroma [L, R], ali je palindrom sa
        // centrom u i_sim takav da nije celokupno smesten u (L, R) - u
        // tom slucaju znamo da je duzina palindroma sa centrom u i bar
        // R-i, a da li je vise od toga, treba proveriti
        d[i] = i <= R ? R - i : 0;
        // prosirujemo palindrom dok god je to moguće krajnji karakteri
        // ^$ obezbedjuju da nije potrebno proveravati granice
        while (t[i - d[i] - 1] == t[i + d[i] + 1])
            d[i]++;
    }

    // ako palindrom sa centrom u i prosiruje desnu granicu onda njega
    // uzimamo za palindrom [L, R] sa centrom u C
    if (i + d[i] > R) {
        C = i;
        R = i + d[i];
    }
}

// pronalazimo najveću duzinu palindroma i pamtimo njegov centar
int maxDuzina = 0, maxCentar = 0;
for (int i = 1; i < t.size() - 1; i++)
    if (d[i] > maxDuzina) {
        maxDuzina = d[i];
        maxCentar = i;
    }

// ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
cout << s.substr((maxCentar - maxDuzina) / 2, maxDuzina) << endl;

return 0;
}
```

Проширивање ниске није неопходно вршити експлицитно.

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

using namespace std;

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

```

// broj pozicija u reci s (pozicije su ili slova originalnih reci,
// ili su izmedju njih). Npr. niska abc ima 7 pozicija i to |a|b|c|
int N = 2 * s.size() + 1;

// d[i] je duzina najduzeg palindroma ciji je centar na poziciji i
// to je ujedno i broj pozicija levo i desno od i na kojima se
// podaci poklapaju - ili su obe pozicije izmedju slova ili su na
// njima jednaka slova
vector<int> d(N);

// Niz d[i] visoko simetrican i da se mnoge njegove vrednosti mogu
// odrediti na osnovu prethodnih, bez potrebe za ponovnim
// izracunavanjem.

// znamo da je [L, R] palindrom sa centrom u C
int C = 0, R = 0; // L = C - (R - C)
for (int i = 0; i < N; i++) {
    // karakter simetrican karakteru i u odnosu na centar C
    int i_sim = C - (i - C);
    if (i < R && i + d[i_sim] < R)
        // nalazimo se unutar palindroma [L, R], ciji je centar C
        // palindrom sa centrom u i_sim i palindrom sa centrom u i su
        // celokupno smesteni u palindrom (L, R)
        d[i] = d[i_sim];
    else {
        // ili se ne nalazimo u okviru nekog prethodnog palindroma ili
        // se nalazimo unutar palindroma [L, R], ali je palindrom sa
        // centrom u i_sim takav da nije celokupno smesten u (L, R) - u
        // tom slucaju znamo da je duzina palindroma sa centrom u i bar
        // R-i, a da li je vise od toga, treba proveriti
        d[i] = i <= R ? R - i : 0;

        // prosirujemo palindrom dok god je to moguće

        // osiguravamo da je i + d[i] stalno paran broj
        if ((i + d[i]) % 2 == 1)
            d[i]++;

        // dok god su pozicije u opsegu i slova na odgovarajucim
        // indeksima jednaka uvecavamo d[i] za 2 (jedno slovo s leva i
        // jedno slovo zdesna)
        while (i - d[i] - 1 >= 0 && i + d[i] + 1 < N &&
            s[(i - d[i] - 1) / 2] == s[(i + d[i] + 1) / 2])
            d[i] += 2; // ukljucujemo dva slova u palindrom
    }

    // ako palindrom sa centrom u i prosiruje desnu granicu onda njega
    // uzimamo za palindrom [L, R] sa centrom u C
    if (i + d[i] > R) {
        C = i;
        R = i + d[i];
    }
}

// pronalazimo najveću duzinu palindroma i pamtimo njegov centar
int maxDuzina = 0, maxCentar = 0;
for (int i = 0; i < N; i++) {

```

```

    if (d[i] > maxDuzina) {
        maxDuzina = d[i];
        maxCentar = i;
    }
}

// ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
int maxPocetak = (maxCentar - maxDuzina) / 2;
cout << s.substr(maxPocetak, maxDuzina) << endl;

return 0;
}

```

Задатак: Најкраћа допуна до палиндрома

Ниска *abaca* није палиндром (не чита се исто слева и сдесна), али ако јој на почетак допишемо карактере *ac*, добијамо ниску *acabaca* која јесте палиндром (чита се исто и слева и сдесна). Напиши програм који за дату ниску одређује дужину најкраћег палиндрома који се може добити дописивањем карактера с леве стране дате ниске.

Улаз: Са стандардног улаза се уноси ниска састављена од N ($1 \leq N \leq 50000$) малих слова енглеске абетеде.

Излаз: На стандардни излаз исписати тражену дужину најкраће проширене ниске која је палиндром.

Пример 1		Пример 2		Пример 3	
Улаз	Излаз	Улаз	Излаз	Улаз	Излаз
abaca	7	anavolimilovana	15	anavolimilovanakarak	25

Решење

Претпоставимо да се најкраћи могући палиндром може добити дописивањем ниске p на ниску s тј. да је ps најкраћи палиндром коме је s суфикс. Тада је $ps = s'p'$, где је s' ниска која се добија обртањем карактера ниске s , док је p' ниска која се добија обртањем карактера ниске p . Дакле, s се завршава са p' и постоји неки њен префикс t такав да је $s = tp'$. Зато је $ptp' = pt'p'$, па је $t = t'$ тј. t мора бити палиндром. Да би дужина префикса p била што мања, дужина палиндрома t мора бити што већа, па је t најдужи могући палиндром којим почиње ниска s .

На пример, да бисмо одредили најдужи палиндром којим се допуњује ниска $s = abacba$, примећујемо да је њен најдужи палиндромски префикс ниска $t = aba$, да је $p' = cba$, па је $p = abc$ и резултујући палиндром је $ps = abcabacba$.

Ако је n_t дужина палиндрома t , а n дужина ниске s , дужина дела p једнака је $n - n_t$, па је укупна дужина траженог палиндрома једнака $(n - n_t) + n = 2n - n_t$.

Палиндром t можемо одредити грубом силом, тако што редом проверавамо све префиксе ниске s и тражимо најдужи палиндром међу њима.

```

#include <iostream>
#include <string>

using namespace std;

bool jePalindrom(const string& s, int n) {
    for (int i = 0, j = n - 1; i < j; i++, j--)
        if (s[i] != s[j])
            return false;
    return true;
}

int duzinaNajduzegPalindromskogPrefiksa(const string& s) {
    for (int n = s.size(); n >= 1; n--)
        if (jePalindrom(s, n))
            return n;
}

```

```

    return 1;
}

int main() {
    string s;
    cin >> s;
    // s razlazemo na prefiks + sufiks tako da je prefiks sto duzi
    // palindrom. Tada je trazeni palindrom (palindrom koji se dobija sa
    // sto manje dopisivanja slova na pocetak niske s) jednak:
    // obrnut_sufiks + prefiks + sufiks
    int duzinaPrefiksa = duzinaNajduzegPalindromskogPrefiksa(s);
    int duzinaSufiksa = s.size() - duzinaPrefiksa;
    int duzinaNajkracegPalindroma = duzinaSufiksa + duzinaPrefiksa + duzinaSufiksa;
    cout << duzinaNajkracegPalindroma << endl;
    return 0;
}

```

Задатак ефикасније можемо решити алгоритмом КМР. Ако уместо ниске s посматрамо ниску ss' добијену надовезивањем ниске добијене обртањем редоследа карактера ниске s најдужи палиндромски префикс ниске s је најдужи префикс ниске ss' који је уједно њен суфикс и који има највише n карактера, где је n дужина ниске s . Да би се спречило да се приликом попуњавања КМР низа урачунају и они суфикси и префикси који садрже целу полазну ниску, довољно је да се између s и s' постави било који карактер који они не садрже (пошто се ниске састоје само од малих слова енглеске абетецеде, можемо, на пример, употребити карактер `.`). Када изградимо ниску $s.s'$, тада на исти начин као у задатку **Префикс суфикс** попуњавамо КМР низ и њен последњи елемент представља дужину најдужег палиндромског префикса.

```

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int duzinaNajduzegPalindromskogPrefiksa(const string& s) {
    string sObratno(s.rbegin(), s.rend());
    string str = s + "." + sObratno;
    vector<int> kmp(str.size() + 1, 0);
    kmp[0] = -1;
    for (int i = 0; i < str.size(); i++) {
        int k = i;
        while (k > 0 && str[i] != str[kmp[k]])
            k = kmp[k];
        kmp[i + 1] = kmp[k] + 1;
    }
    return kmp[kmp.size() - 1];
}

int main() {
    string s;
    cin >> s;
    // s razlazemo na prefiks + sufiks tako da je prefiks sto duzi
    // palindrom. Tada je trazeni palindrom dobijen sa sto manje
    // dopisivanja slova na pocetak jednak:
    // obrni(sufiks) + prefiks + sufiks
    int duzinaPrefiksa = duzinaNajduzegPalindromskogPrefiksa(s);
    int duzinaSufiksa = s.size() - duzinaPrefiksa;
    int duzinaNajkracegPalindroma = duzinaSufiksa + duzinaPrefiksa + duzinaSufiksa;
    cout << duzinaNajkracegPalindroma << 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 абетецеде, чија је дужина највише 5000 карактера.

Издаз: На стандардни издаз исписати само тражену дужину најдужег палиндромског подниза.

Пример

Улаз	Издаз
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 разматрамо 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.

$$\begin{aligned}f(l, d) &= 0, & \text{za } l > d \\f(l, d) &= 1, & \text{za } l = d \\f(l, d) &= 2 + f(l + 1, d - 1), & \text{za } l < d \text{ i } s_l = s_d \\f(l, d) &= \max(f(l + 1, d), f(l, d - 1)), & \text{za } l < d \text{ i } s_l \neq s_d\end{aligned}$$

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 дијагонали по дијагонали. На дијагонали испод главне уписујемо све нуле, на главну дијагонали све јединице, а затим попуњавамо једну по једну дијагонали 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 променљиву бележимо одговарајући елемент претходне дијагонале, на његово место уписујемо одговарајући елемент текуће дијагонале, а онда на место тог елемента уписујемо вредност привремене променљиве 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;
}
```

3.3 Регуларни изрази

Задатак: Датуми

У тексту се налази одређен број датума, али су записани у два различита формата. Неки су записани у облику `dd.mm.gggg`. (при чему дан и месец могу бити и једноцифрени), а неки у облику `mm/dd/gggg` (ту су дан и месец увек двоцифрени). Напиши програм који издваја све те датуме и све их исписује у првом формату, са двоцифреним даном и месецом. Претпоставити да су сви датуми у документу исправно записани.

Улаз: Са стандардног улаза се уноси линија по линија текста, све до краја улаза. Претпоставити да се у свакој линији налази највише један датум.

Издаз: На стандардни издаз исписати тражене датуме, један испод другог.

Пример

Улаз	Издаз
Nikola Tesla se rodio 10.7.1856.	10.07.1856.
Nikola Tesla was born on 07/10/1856.	10.07.1856.

Решење

Најједноставнији начин да се задатак реши је да се употребе регуларни изрази. Регуларним изразима се описују шаблони текста на основу којих можемо вршити претрагу и замену. Основна врста регуларног израза је ниска. На пример, регуларни израз је `abc` и једина ниска која је тим изразом покривена је управо `abc`. Регуларни изрази се комбинују следећим операторима.

- `|` - овај оператор се чита “или”. На пример, регуларни израз `abc|def` је шаблон у који се уклапају две могуће ниске: прва је `abc`, а друга `def`.
- `*` - овај оператор се чита “нула или више пута”. На пример, регуларни израз `a*` означава да се `a` јавља нула или више пута и обухвата празну ниску, затим `a`, `aa`, `aaa` и тако даље.
- `+` - овај оператор се чита “један или више пута”. На пример, регуларни израз `a+` означава да се `a` јавља један или више пута и обухвата ниске `a`, `aa`, `aaa` и тако даље.
- `?` - овај оператор се чита “нула или један пут”. На пример, регуларни израз `a?` описује или празну ниску или ниску `a`.
- Када се на израз допише `{n}` то означава да се ниска описана изразом понавља тачно `n` пута. На пример, `a{5}` означава ниску `aaaaa`.
- регуларни изрази постављени један поред другог представљају дописивање ниски. На пример, `(ab)+(cd)?` означава да се `ab` јавља један или више пута, за чим може, а не мора да следи `cd`. Неке ниске које су описане овим изразом су `ab`, `abcd`, `abab`, `ababcd` итд.
- Изрази облика `[a-z]` су такозване карактерске класе и представљају један карактер из датог распона. Израз `[a-z]` описује ниске `a`, `b`, `c` итд., док израз `[0-9]` описује једну цифру.
- Израз `.` описује било који карактер, осим преласка у нови ред.

Постоји још велики број регуларних операција и начина да се регуларни изрази компактније запишу, али их нећемо помињати.

Датуми првог облика се могу описати регуларним изразом `[0-9][0-9]?[.][0-9][0-9]?[.][0-9]{5}`, а другог облика изразом `[0-9]{2}/[0-9]{2}/[0-9]{4}`.

Подршка за регуларне изразе у C++-у дата је кроз функције заглавља <regex>.

Функција `regex_search` проверава да ли се неки део дате ниске уклапа у шаблон описан датим регуларним изразом. Резултати претраге уписују се у објекат типа `smatch`. У питању је колекција чијим се појединачним елементима може приступити коришћењем индексног оператора `[]`.

Функција `regex_match` проверава да ли се цела дата ниска уклапа у шаблон описан датим регуларним изразом.

```
#include <iostream>
#include <regex>

using namespace std;

string pad(string s) {
    return s.size() == 1 ? ("0" + s) : s;
}

int main() {
    string format1 = "([0-9][0-9]?)([0-9][0-9]?)([0-9][0-9][0-9][0-9])([0-9])";
    string format2 = "([0-9]{2})/([0-9]{2})/([0-9]{4})";
    regex datum("(" + format1 + "|" + format2 + ")");
    smatch sm;
    string linija;
    while (getline(cin, linija))
        if (regex_search(linija, sm, datum)) {
            string prepoznat_datum = sm[0];
            if (regex_match(prepoznat_datum, sm, regex(format1)))
                cout << pad(sm[1]) << "." << pad(sm[2]) << "." << sm[3] << "." << endl;
            else if (regex_match(prepoznat_datum, sm, regex(format2)))
                cout << sm[2] << "." << sm[1] << "." << sm[3] << "." << endl;
        }
    return 0;
}
```

3.4 Формалне граматике

Задатак: Превођење потпуно заграђеног изрази 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
<code>((3*5)+(7+(2*1)))*4</code>	<code>35*721*++4*</code>

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 сисџи* 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 операнда).

Променљива i мења своју вредност кроз рекурзивне позиве. Стога ћемо је преноси 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 изрази 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;
}
```

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 преводјење 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: 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 граматичи придружимо 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 _predvid;

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 množимо 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 алгорита 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, množимо 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 množимо текући сабирак са 4 (добијамо вредност 4), затим читамо 5 и množимо га са 5 (добијамо вредност 20) и затим читамо 2 и množимо 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 množимо вредност текућег сабирка (добијамо вредност 7), затим читамо 2 и њоме množимо 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;
    // zavrшили smo sa obradom tekuceg broja i pripremamo se za
    // citanje narednog
    tekuciBroj = 0;

    // ako smo stigli do kraja ili procitali aditivni operator
    // zavrшили 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()) {
            // pripremamo 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 изразица 3, затим $3+4$, затим $3+4-2$, затим $3+4-2+5$ и на крају $3+4-2+5*7$. Вредност претходно израчунатог изразица помаже да се израчуна вредност наредног изразица. На пример, ако знамо вредност изразица 3, тада вредност изразица $3+4$ добијамо тако што на ту вредност додајемо вредност броја 4.

У општем случају, ако знамо вредност изразица e , тада вредност изразица $e+x$ добијамо тако што на ту вредност додамо вредност броја x . Слично, ако знамо вредност изразица e , тада вредност изразица $e-x$ добијамо тако што од те вредности одузмемо вредност броја x .

Случај множења је мало компликованији.

Ако знамо вредност изразица $3+4-2+5$, вредност изразица $3+4-2+5*7$ 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 изразица $3+4-2$), затим је помножити са 7 и на крају тај производ додати на вредност изразица.

У општем случају, ако имамо израз облика $e+e'*x$, и знамо вредност изразица $e+e'$, вредност новог изразица добијамо тако што од вредности изразица $e+e'$ одузмемо вредност e' а затим додамо вредност $e'*x$. Да би ово било могуће уз вредност сваког текућег изразица 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 azuירarani poslednji sabirak,
                    // nadajuci se da se iza njega nece vise javljati znak *
                    rezultat += poslednjiSabirak;
                    break;
            }
            op = s[i];
        }
}
```

```
    }  
    // zavrшили 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).

Издаз: 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	Издаз
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 изводи 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;
}

```

Израчунавање вредности израза је опет комбинација преводјења 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 algorithm*, 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 Т и споредни колосек је усправна црта). Делови израза прелазе са десног на леви крај (замислимо да иду по горњој ивици слова Т). 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.

----- (2+3)*5+2*5	----- 2+3)*5+2*5	2 ----- +3)*5+2*5
	(	(
2 ----- 3)*5+2*5	23 -----)*5+2*5	23+ ----- *5+2*5
+	+	
(	(	
23+ ----- 5+2*5	23+5 ----- +2*5	23+5* ----- 2*5
*	*	+
23+5*2 ----- *5	23+5*2 ----- 5	23+5*25 -----
	*	*
+	+	+
23+5*25*+ -----		

```

#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) {
        // pokušavamo da izračunamo vrednost izraza
        int rez;
        if (vrednost(s, rez))
            cout << rez << endl;
        else
            cout << "deljenje nulom" << endl;
    }
}
```


Глава 4

Геометријски алгоритми

4.1 Скаларни и векторски производ

Задатак: Колинеарност и нормалност вектора

Напиши програм који одређује да ли су два тродимензионална вектора међусобно колинеарна и да ли су међусобно нормална.

Улаз: Прва линија стандардног улаза садржи три цела броја између -100 и 100 раздвојена са по једним размаком, које представљају координате првог вектора. Друга линија стандардног улаза садржи три цела броја између -100 и 100 раздвојена са по једним размаком, које представљају координате другог вектора.

Издаз: На стандардни издаз исписати текст `kolinearni` ако су вектори колинеарни, `normalni` ако су нормални или - ако нису ни једно ни друго.

Пример 1

Улаз Издаз
3 2 5 normalni
1 -4 1

Пример 2

Улаз Издаз
3 2 5 kolinearni
6 4 10

Решење

Два вектора су колинеарна ако и само ако им је векторски производ једнак нули, а нормална ако и само ако им је скаларни производ једнак нули. Векторски производ вектора (x_1, y_1, z_1) и (x_2, y_2, z_2) се може израчунати помоћу детерминанте

$$\begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix}$$

и једнак је вектору $(y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$. Њихов скаларни производ једнак је $x_1x_2 + y_1y_2 + z_1z_2$.

```
#include <iostream>
```

```
using namespace std;
```

```
int skalarni_proizvod(int x1, int y1, int z1,  
                     int x2, int y2, int z2) {  
    return x1*x2 + y1*y2 + z1*z2;  
}
```

```
void vektorski_proizvod(int x1, int y1, int z1,  
                       int x2, int y2, int z2,  
                       int& x, int& y, int& z) {  
    // izracunavamo determinantu:
```

```
// i j k
// x1 y1 z1
// x2 y2 z2
x = y1*z2 - y2*z1;
y = z1*x2 - x1*z2;
z = x1*y2 - x2*y1;
}

int main() {
    int x1, y1, z1;
    cin >> x1 >> y1 >> z1;
    int x2, y2, z2;
    cin >> x2 >> y2 >> z2;

    bool specijalni = false;

    if (skalarni_proizvod(x1, y1, z1, x2, y2, z2) == 0) {
        cout << "normalni" << endl;
        specijalni = true;
    }

    int x, y, z;
    vektorski_proizvod(x1, y1, z1, x2, y2, z2, x, y, z);
    if (x == 0 && y == 0 && z == 0) {
        cout << "kolinearni" << endl;
        specijalni = true;
    }

    if (!specijalni)
        cout << "-" << endl;

    return 0;
}
```

Задатак: Колинеарне тачке

Напиши програм који одређује колико тачака из датог скупа тачака припада датој правој.

Улаз: Са стандардног улаза се учитавају координате две различите тачке којима је одређена права (у првој линији се налазе два цела броја раздвојена размаком које представљају координате прве тачке, а у другој два цела броја раздвојена размаком које представљају координате друге тачке). Након тога се учитава број тачака n ($1 \leq n \leq 100$), а затим из наредних n линија координате тих тачака (свака линија садржи два цела броја раздвојена размаком).

Издаз: На стандардни издаз исписати тражени број тачака које припадају правој.

Пример

Улаз	Издаз
1 2	3
3 4	
5	
-8 -7	
-8 -9	
1 2	
2 3	
0 0	

Решење

Тачке $A = (x_1, y_1)$, $B = (x_2, y_2)$ и $C = (x_3, y_3)$ су колинеарне ако и само ако су вектори $\vec{AB} = (x_2 - x_1, y_2 - y_1, 0)$ и $\vec{AC} = (x_3 - x_1, y_3 - y_1, 0)$ колинеарни, тј. ако им је векторски производ 0. До овог закључка се може

доћи и ако се примети да су тачке коолинеарне ако и само ако је површина троугла који образују једнака нули (а површина троугла једнака је половини интензитета векторског производа).

```
#include <iostream>

using namespace std;

void vektorski_proizvod(int x1, int y1, int z1,
                      int x2, int y2, int z2,
                      int& x, int& y, int& z) {
    // izracunavamo determinantu:
    // i j k
    // x1 y1 z1
    // x2 y2 z2
    x = y1*z2 - y2*z1;
    y = z1*x2 - x1*z2;
    z = x1*y2 - x2*y1;
}

bool kolinearni_vektori(int x1, int y1, int z1,
                      int x2, int y2, int z2) {
    int x, y, z;
    vektorski_proizvod(x1, y1, z1,
                      x2, y2, z2,
                      x, y, z);
    return x == 0 && y == 0 && z == 0;
}

bool kolinearne_tacke(int x1, int y1, int x2, int y2, int x3, int y3) {
    return kolinearni_vektori(x1 - x2, y1 - y2, 0,
                             x1 - x3, y1 - y3, 0);
}

int main() {
    int x1, y1;
    cin >> x1 >> y1;
    int x2, y2;
    cin >> x2 >> y2;
    int n;
    cin >> n;
    int broj = 0;
    for (int i = 0; i < n; i++) {
        int x, y;
        cin >> x >> y;
        if (kolinearne_tacke(x1, y1, x2, y2, x, y))
            broj++;
    }
    cout << broj << endl;
    return 0;
}
```

Пошто претпостављамо да вектори $\vec{AB}$ и $\vec{AC}$ припадају равни xOy , њихов векторски производ је ортогоналан на ту раван. Заиста, његове координате су $((x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1), 0, 0)$ и он је једнак нули ако и само ако је $(x_2 - x_1)(y_3 - y_1) = (y_2 - y_1)(x_3 - x_1)$.

```
#include <iostream>

using namespace std;

bool kolinearne_tacke(int x1, int y1, int x2, int y2, int x3, int y3) {
```

```

    return (x1-x2)*(y1-y3) == (y1-y2)*(x1-x3);
}

int main() {
    int x1, y1;
    cin >> x1 >> y1;
    int x2, y2;
    cin >> x2 >> y2;
    int n;
    cin >> n;
    int broj = 0;
    for (int i = 0; i < n; i++) {
        int x, y;
        cin >> x >> y;
        if (kolinearne_tacke(x1, y1, x2, y2, x, y))
            broj++;
    }
    cout << broj << endl;
    return 0;
}

```

Задатак: Са исте стране

Напиши програм који утврђује колико се датих тачака налази са исте стране праве као и једна задатака тачка.

Улаз: Са стандардног улаза се уносе две различите тачке A и B које одређују праву, затим тачка T , након тога број тачака n ($10 \leq n \leq 100$), и након тога n тачака. Свака тачка је задата у посебном реду, помоћу своје две целобројне координате раздвојене једним размаком.

Издаз: Опис излазних података.

Пример

Улаз	Издаз
1 1	3
2 2	
0 1	
5	
3 4	
7 5	
2 6	
-1 -5	
-5 -2	

Објашњење

Тачке $(3, 4)$, $(2, 6)$ и $(-5, -2)$ се налазе са исте стране праве одређене тачкама $(1, 1)$ и $(2, 2)$ као и тачка $(0, 1)$.

Решење

Тачка T_1 и T_2 су са исте стране праве AB ако и само ако је оријентација тројки ABT_1 и ABT_2 једнака. Оријентација тројке ABT се може одредити на основу знака векторског производа вектора AT и BT .

```
#include <iostream>
```

```
using namespace std;
```

```
// moguće orijentacije trojke tacaka
```

```
enum Orijentacija {POZITIVNA, KOLINEARNE, NEGATIVNA};
```

```
Orijentacija orijentacija(int xa, int ya, int xb, int yb, int xc, int yc) {
    int d = (xb-xa)*(yc-ya) - (xc-xa)*(yb-ya);
    if (d > 0)
```

```

    return POZITIVNA;
else if (d < 0)
    return NEGATIVNA;
else
    return KOLINEARNE;
}

bool saIsteStrane(int xa, int ya, int xb, int yb,
                 int x1, int y1, int x2, int y2) {
    return orijentacija(xa, ya, xb, yb, x1, y1) ==
           orijentacija(xa, ya, xb, yb, x2, y2);
}

int main() {
    int xa, ya;
    cin >> xa >> ya;
    int xb, yb;
    cin >> xb >> yb;
    int x1, y1;
    cin >> x1 >> y1;
    int n;
    cin >> n;
    int broj = 0;
    for (int i = 0; i < n; i++) {
        int x2, y2;
        cin >> x2 >> y2;
        if (saIsteStrane(xa, ya, xb, yb, x1, y1, x2, y2))
            broj++;
    }
    cout << broj << endl;
    return 0;
}

```

Задатак: Тачка у троуглу

Напиши програм који проверава да ли се тачка налази у унутрашњости троугла.

Улаз: Са стандардног улаза се уноси 8 реалних бројева из интервала $[-10, 10]$, заокружених на две децимале. У првом реду се уносе x и y координата тачке која се анализира, а у наредна три реда x и y координате темена троугла. Међу 4 унете тачке нема колинеарних.

Излаз: На стандардни излаз исписати да ако тачка припада троуглу или не ако не припада.

Пример 1

Улаз	Излаз
0.00 2.00	da
-3.00 -3.00	
-1.00 4.00	
3.00 2.00	

Пример 2

Улаз	Излаз
-3.50 3.50	ne
-3.00 -3.00	
-1.00 4.00	
3.00 2.00	

Решење

Тачка T је унутар троугла ABC ако и само ако је површина троугла ABC једнака збиру површина троуглова ABT , ATC и TBC .

```

#include <iostream>
#include <cmath>

using namespace std;

// tolerancija
const double EPS = 1e-6;

```

```
// tacka je zadata svojim dvema koordinatama
struct Tacka {
    double x, y;
    Tacka(double x_, double y_) {
        x = x_; y = y_;
    }
};

double povrsinaTrougla(const Tacka& A, const Tacka& B, const Tacka& C) {
    // pertle (cik-cak)
    return abs(+ A.x*B.y + B.x*C.y + C.x*A.y
               - A.x*C.y - C.x*B.y - B.x*A.y) / 2.0;
}

bool tackaUTrouglu(const Tacka& T,
                   const Tacka& A, const Tacka& B, const Tacka& C) {
    // racunamo povrsinu trougla ABC
    double P = povrsinaTrougla(A, B, C);
    // racunamo povrsine trouglova TBC, TAC i TAB
    double P1 = povrsinaTrougla(T, B, C);
    double P2 = povrsinaTrougla(A, T, C);
    double P3 = povrsinaTrougla(A, B, T);
    // proveravamo da li one u zbiru daju povrsinu trougla ABC
    // poredimo dve realne vrednosti na jednakost
    return abs(P - (P1 + P2 + P3)) < EPS;
}

int main() {
    double x, y;
    cin >> x >> y;
    Tacka P(x, y);
    cin >> x >> y;
    Tacka A(x, y);
    cin >> x >> y;
    Tacka B(x, y);
    cin >> x >> y;
    Tacka C(x, y);
    if (tackaUTrouglu(P, A, B, C))
        cout << "da" << endl;
    else
        cout << "ne" << endl;
    return 0;
}
```

Тачка T је унутар троугла ABC ако и само ако је тачка T са исте стране праве AB као и тачка C , са исте стране праве BC као и тачка A и са исте стране тачке AC као и тачка B тј. ако сви троуглови ABT , BCT и CAT имају исту оријентацију.

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
```

```
// tolerancija
const double EPS = 1e-6;
```

```
// tacka je zadata svojim dvema koordinatama
```

```

struct Tacka {
    double x, y;
    Tacka(double x_, double y_) {
        x = x_; y = y_;
    }
};

// moguće orijentacije trojke tacaka
enum Orijentacija {POZITIVNA, KOLINEARNE, NEGATIVNA};

Orijentacija orijentacija(const Tacka& A, const Tacka& B, const Tacka& C) {
    double d = (B.x-A.x)*(C.y-A.y) - (C.x-A.x)*(B.y-A.y);
    if (abs(d) < EPS)
        return KOLINEARNE;
    else if (d > EPS)
        return POZITIVNA;
    else
        return NEGATIVNA;
}

bool tackaUTrouglu(const Tacka& T,
                   const Tacka& A, const Tacka& B, const Tacka& C) {
    Orijentacija o1 = orijentacija(A, B, T);
    Orijentacija o2 = orijentacija(B, C, T);
    Orijentacija o3 = orijentacija(C, A, T);
    return o1 == o2 && o2 == o3;
}

int main() {
    double x, y;
    cin >> x >> y;
    Tacka P(x, y);
    cin >> x >> y;
    Tacka A(x, y);
    cin >> x >> y;
    Tacka B(x, y);
    cin >> x >> y;
    Tacka C(x, y);
    if (tackaUTrouglu(P, A, B, C))
        cout << "da" << endl;
    else
        cout << "ne" << endl;
    return 0;
}

```

Задатак: Пресек дужи

Напиши програм који одређује да ли се две дужи секу и ако се секу у једној тачки, координате те тачке.

Улаз: Четири линије стандардног улаза садрже координате четири различите тачке (у свакој линији наведена су два цела броја између -10 и 10, раздвојена размаком).

Издаз: У првој линији стандардног излаза исписати *да* ако се дужи секу тј. *не* у супротном. Ако се дужи секу тачно у једној тачки, у другој линији исписати њене координате (два реална броја заокружена на пет децимала, раздвојене размаком).

Пример 1

Улаз	Израз
0 0	da
1 1	0.50000 0.50000
0 1	
1 0	

Пример 2

Улаз	Израз
0 0	da
5 0	3.00000 0.00000
3 0	
3 5	

Пример 3

Улаз	Израз
0 2	ne
2 0	
-2 -2	
0 0	

Решење

Један начин да се задатак реши је да се провери да ли су тачке A_1 и A_2 са разне стране праве одређене тачкама B_1 и B_2 као и да ли су тачке B_1 и B_2 са разних страна праве одређене тачкама A_1 и A_2 . Ако јесу, тада постоји јединствен пресек. Потребно је додатно обратити пажњу на мноштво дегенерисаних случајева који могу наступити (они се идентификују испитивањем колинеарности тачака A_1 , A_2 , B_1 и B_2).

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

using namespace std;

// moguće orijentacije trojke tacaka
enum Orijentacija {POZITIVNA, KOLINEARNE, NEGATIVNA};

Orijentacija orijentacija(int xa, int ya, int xb, int yb, int xc, int yc) {
    int d = (xb-xa)*(yc-ya) - (xc-xa)*(yb-ya);
    if (d > 0)
        return POZITIVNA;
    else if (d < 0)
        return NEGATIVNA;
    else
        return KOLINEARNE;
}

bool kolinearne(int x1, int y1, int x2, int y2, int x3, int y3) {
    return orijentacija(x1, y1, x2, y2, x3, y3) == KOLINEARNE;
}

// za tri kolinearne tacke proverava da li tacka (x, y) pripada duzi
// (x1, y1) -- (x2, y2)
bool pripadaDuzi(int x1, int y1, int x2, int y2, int x, int y) {
    return min(x1, x2) <= x && x <= max(x1, x2) &&
        min(y1, y2) <= y && y <= max(y1, y2);
}

bool saRaznihStrana(int xa, int ya, int xb, int yb,
                    int x1, int y1, int x2, int y2) {
    return orijentacija(xa, ya, xb, yb, x1, y1) !=
        orijentacija(xa, ya, xb, yb, x2, y2);
}

int main() {
    int xa1, ya1, xa2, ya2;
    cin >> xa1 >> ya1 >> xa2 >> ya2;
    int xb1, yb1, xb2, yb2;
    cin >> xb1 >> yb1 >> xb2 >> yb2;

    bool sekuSe;
    bool jedinstvenPresek;
    double x, y;

    if (kolinearne(xa1, ya1, xa2, ya2, xb1, yb1) &&
```



```

    kolinearne(xa1, ya1, xa2, ya2, xb2, yb2)) {
if (pripadaDuzi(xa1, ya1, xa2, ya2, xb1, yb1)) {
    sekuSe = true;
    if (pripadaDuzi(xa1, ya1, xa2, ya2, xb2, yb2))
        jedinstvenPresek = false;
    else {
        if ((xb1 == xa1 && yb1 == ya1) || (xb1 == xa2 && yb1 == ya2)) {
            jedinstvenPresek = true;
            x = xb1; y = yb1;
        } else {
            jedinstvenPresek = false;
        }
    }
}
} else if (pripadaDuzi(xa1, ya1, xa2, ya2, xb2, yb2)) {
    sekuSe = true;
    if ((xb2 == xa1 && yb2 == ya1) || (xb2 == xa2 && yb2 == ya2)) {
        jedinstvenPresek = true;
        x = xb2; y = yb2;
    } else {
        jedinstvenPresek = false;
    }
} else
    sekuSe = false;
} else if (kolinearne(xa1, ya1, xa2, ya2, xb1, yb1)) {
    if (pripadaDuzi(xa1, ya1, xa2, ya2, xb1, yb1)) {
        sekuSe = true;
        jedinstvenPresek = true;
        x = xb1; y = yb1;
    } else
        sekuSe = false;
} else if (kolinearne(xa1, ya1, xa2, ya2, xb2, yb2)) {
    if (pripadaDuzi(xa1, ya1, xa2, ya2, xb2, yb2)) {
        sekuSe = true;
        jedinstvenPresek = true;
        x = xb2; y = yb2;
    } else
        sekuSe = false;
} else {
    if (saRaznihStrana(xa1, ya1, xa2, ya2, xb1, yb1, xb2, yb2) &&
        saRaznihStrana(xb1, yb1, xb2, yb2, xa1, ya1, xa2, ya2)) {
        sekuSe = true;
        jedinstvenPresek = true;
        int imenilac = (xa1-xa2)*(yb1-yb2)-(ya1-ya2)*(xb1-xb2);
        int brojilacX = (xa1*ya2-ya1*xa2)*(xb1-xb2)-(xa1-xa2)*(xb1*yb2-yb1*xb2);
        int brojilacY = (xa1*ya2-ya1*xa2)*(yb1-yb2)-(ya1-ya2)*(xb1*yb2-yb1*xb2);
        x = (double)brojilacX / (double)imenilac;
        y = (double)brojilacY / (double)imenilac;
    } else
        sekuSe = false;
}
}
if (sekuSe) {
    cout << "da" << endl;
    if (jedinstvenPresek)
        cout << fixed << showpoint << setprecision(5)
            << x << " " << y << endl;
} else {
    cout << "ne" << endl;
}

```

```

    }

    return 0;
}

```

Задатак можемо решити и применом параметарске једначине дужи. Дуж A_1A_2 има параметарску једначину $\vec{A_1} + t_a \vec{A_1A_2}$, за $0 \leq t_a \leq 1$, док дуж B_1B_2 има параметарску једначину $\vec{B_1} + t_b \vec{B_1B_2}$, за $0 \leq t_b \leq 1$. Одавде се добија систем једначина

$$\begin{aligned}(x_{A_2} - x_{A_1})t_a + (x_{B_1} - x_{B_2})t_b &= x_{B_1} - x_{A_1} \\ (y_{A_2} - y_{A_1})t_a + (y_{B_1} - y_{B_2})t_b &= y_{B_1} - y_{A_1}\end{aligned}$$

Ако је детерминанта система различита од нуле, тада систем има јединствено решење и оно одређује тачку пресека две праве. Да би се проверило да је то уједно пресек дужи, потребно је проверити да ли оба израчуната параметра припадају интервалу $[0, 1]$.

Ако је детерминанта система једнака нули, онда су праве паралелне, па проверавамо дегенерисане случајеве (могуће је да су праве било паралелне, било да се поклапају).

```

#include <iostream>
#include <iomanip>

using namespace std;

enum BrojResenja {NEMA_RESENJA, JEDINSTVENO_RESENJE, BESKONACNO_RESENJA};

BrojResenja resiSistem(int a11, int a12, int a21, int a22, int b1, int b2,
                       double& x1, double& x2) {
    int D = a11*a22 - a12*a21;
    int D1 = b1*a22 - a12*b2;
    int D2 = a11*b2 - a21*b1;
    if (D == 0) {
        if (D1 == 0 && D2 == 0)
            return BESKONACNO_RESENJA;
        else
            return NEMA_RESENJA;
    }
    x1 = (double)D1/(double)D;
    x2 = (double)D2/(double)D;
    return JEDINSTVENO_RESENJE;
}

// za tri kolinearne tacke proverava da li tacka (x, y) pripada duzi
// (x1, y1) -- (x2, y2)
bool pripadaDuzi(int x1, int y1, int x2, int y2, int x, int y) {
    return min(x1, x2) <= x && x <= max(x1, x2) &&
           min(y1, y2) <= y && y <= max(y1, y2);
}

int main() {
    int xa1, ya1, xa2, ya2;
    cin >> xa1 >> ya1 >> xa2 >> ya2;
    int xb1, yb1, xb2, yb2;
    cin >> xb1 >> yb1 >> xb2 >> yb2;

    bool sekuSe;
    bool jedinstvenPresek;
    double x, y;

```

```

int a11 = xa2-xa1, a12 = xb1-xb2, b1 = xb1-xa1;
int a21 = ya2-ya1, a22 = yb1-yb2, b2 = yb1-ya1;
double t1, t2;
BrojResenja broj = resiSistem(a11, a12, a21, a22, b1, b2, t1, t2);
if (broj == JEDINSTVENO_RESENJE) {
    if (0 <= t1 && t1 <= 1 && 0 <= t2 && t2 <= 1) {
        sekuSe = true;
        jedinstvenPresek = true;
        x = xa1 + t1*(xa2-xa1);
        y = ya1 + t1*(ya2-ya1);
    } else {
        sekuSe = false;
    }
} else if (broj == NEMA_RESENJA) {
    sekuSe = false;
} else {
    if (pripadaDuzi(xa1, ya1, xa2, ya2, xb1, yb1) &&
        pripadaDuzi(xa1, ya1, xa2, ya2, xb2, yb2)) {
        sekuSe = true;
        jedinstvenPresek = false;
    } else if (pripadaDuzi(xa1, ya1, xa2, ya2, xb1, yb1)) {
        sekuSe = true;
        if (xb1 == xa1 || xb1 == xa2) {
            jedinstvenPresek = true;
            x = xb1; y = yb1;
        } else {
            jedinstvenPresek = false;
        }
    } else if (pripadaDuzi(xa1, ya1, xa2, ya2, xb2, yb2)) {
        sekuSe = true;
        if (xb2 == xa1 || xb2 == xa2) {
            jedinstvenPresek = true;
            x = xb2; y = yb2;
        } else {
            jedinstvenPresek = false;
        }
    } else {
        sekuSe = false;
    }
}

if (sekuSe) {
    cout << "da" << endl;
    if (jedinstvenPresek)
        cout << fixed << showpoint << setprecision(5)
            << x << " " << y << endl;
} else {
    cout << "ne" << endl;
}

return 0;
}

```

4.2 Многоуглови

Задатак: Конструкција простог многоугла

Прост многоугао је многоугао у ком се никоје две стране не секу (осим што се суседне стране додирују у заједничком темену). Напиши програм који за дати скуп тачака (у ком нису све тачке колинеарне) одређује неки прост многоугао ком је скуп темена једнак том скупу тачака.

Улаз: Са стандардног улаза се учитава број n ($3 \leq n \leq 50000$), а затим и n тачака (свака тачка је описана у посебном реду помоћу своје две целобројне координате раздвојене размаком). Све уčitане тачке су различите и нису све колинеарне.

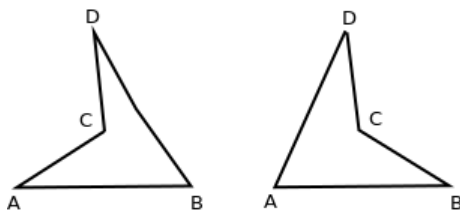
Изназ: На стандардни излаз исписати пермутацију учитаног низа тачака која представља редослед темена конструисаног простог многоугла (тачке треба да буду описане на исти начин као и на улазу).

Пример

Улаз	Изназ
5	5 1
3 1	5 2
0 4	2 3
5 1	0 4
2 3	3 1
5 2	

Решење

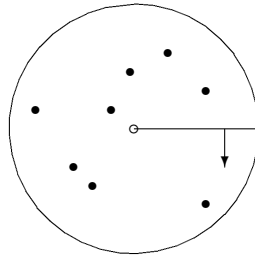
Некада је за дати скуп тачака у равни могуће конструисати више различитих простих многоуглова, односно решење није једнозначно одређено (слика 4.1).



Слика 4.1: Тачке A , B , C и D и два различита проста многоугла која оне одређују.

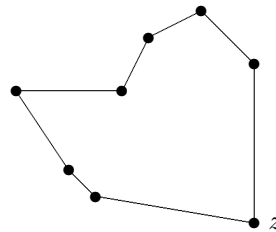
Нека је C неки круг, чија унутрашњост садржи све тачке. За налажење таквог круга довољно је израчунати највеће међу растојањима произвољне тачке равни (центра круга) до свих осталих тачака – сложеност овог корака је $O(n)$. Површина C може се “пребрисати” (прегледати) ротирајућом полуправом којој је почетак центар C (слика 4.2). Претпоставимо за тренутак да ротирајућа права у сваком тренутку садржи највише једну тачку. Очекујемо да ћемо спајањем тачака оним редом којим полуправа наилази на њих добити прост многоугао. Докажимо то. Означимо тачке, уређене у складу са редоследом наилаaska полуправе на њих, са $P_1, P_2, \dots, P_n$ (прва тачка бира се произвољно). За свако i , $1 \leq i \leq n$, страница $P_i P_{i-1}$ (односно $P_1 P_n$ за $i = 1$) садржана је у новом (дисјунктном) исечку круга, па се не сече ни са једном другом страницом. Ако би ово тврђење било тачно, добијени многоугао би морао да буде прост. Међутим, угао између полуправих кроз неке две узастопне тачке P_i и P_{i+1} може да буде већи од π . Тада исечак који садржи дуж $P_i P_{i+1}$ садржи више од пола круга и није конвексна фигура, а дуж $P_i P_{i+1}$ пролази кроз друге исечке круга, па може да сече друге странице многоугла. Да бисмо се уверили да је то могуће, довољно је да уместо нацртаног замислимо круг са центром ван круга са слике 4.2. Ово је пример специјалних случајева на које се наилази при решавању геометријских проблема.

Да би се решио уочени проблем, могу се, на пример, фиксирати произвољне три неколинеарне тачке из скупа, а за центар круга изабрати нека тачка унутар њима одређеног троугла (на пример тежиште, које се лако налази). Овакав избор гарантује да ни један од добијених сектора круга неће имати угао већи од π . Затим сортирамо тачке према положају у кругу са центром z . Прецизније, сортирају се углови између x -осе и полуправих од z ка осталим тачкама. Ако две или више тачака заклапају исти угао са x -осом, оне се даље сортирају растуће према растојању од тачке z . На крају, тачке повезујемо у складу са добијеним уређењем,



Слика 4.2: Пролазак тачака у кругу ротирајућом полуправом.

по две узатопне. Пошто све тачке леже лево од тачке z , до дегенерисаног случаја о коме је било речи не може доћи. Прост многоугао добијен овим поступком за тачке са слике 4.2 приказан је на слици 4.3. Основна компонента временске сложености овог алгорита потиче од сортирања. Сложеност алгорита је дакле $O(n \log n)$.



Слика 4.3: Конструкција простог многоугла.

Угао φ који права $y = mx + b$ заклапа са x осом добија се коришћењем везе $m = \tan \varphi$, односно из једначине $\varphi = \arctan m$. Међутим, углови се не морају експлицитно израчунавати. Углови се користе само за налажење редоследа којим треба повезати тачке и исти редослед добија се уређењем нагиба одговарајућих полуправих; то чини непотребним израчунавање аркустангенса. Из истог разлога непотребно је израчунавање растојања кад две тачке имају исти нагиб — довољно је израчунати квадрате растојања. Дакле, нема потребе за израчунавањем квадратних коренова.

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

using namespace std;

// tacka je zadata svojim dvema koordinatama
struct Tacka {
    int x, y;
    Tacka(int x_ = 0, int y_ = 0) {
        x = x_; y = y_;
    }
};

bool kolinearne(const Tacka& t1, const Tacka& t2, const Tacka& t3) {
    return (t1.x-t2.x)*(t1.y-t3.y) == (t1.y-t2.y)*(t1.x-t3.x);
}

double kvadratRastojanja(double x1, double y1, double x2, double y2) {
    double dx = x1 - x2, dy = y1 - y2;
    return dx*dx + dy*dy;
}
```

```

void prostMnogougao(vector<Tacka>& tacke) {
    int i = 2;
    while (kolinearne(tacke[0], tacke[1], tacke[i]))
        i++;
    double x0 = (tacke[0].x + tacke[1].x + tacke[i].x) / 3.0;
    double y0 = (tacke[0].y + tacke[1].y + tacke[i].y) / 3.0;
    sort(begin(tacke), end(tacke),
        [x0, y0](const Tacka& t1, const Tacka& t2) {
            double x1 = t1.x - x0, y1 = t1.y - y0;
            double x2 = t2.x - x0, y2 = t2.y - y0;
            double ugao1 = atan2(y1, x1);
            double ugao2 = atan2(y2, x2);
            const double EPS = 1e-12;
            if (ugao1 < ugao2 - EPS) {
                return true;
            }
            if (ugao2 < ugao1 - EPS) {
                return false;
            }
            return kvadratRastojanja(x0, y0, x1, y1) <
                kvadratRastojanja(x0, y0, x2, y2);
        });
}

int main() {
    int n;
    cin >> n;
    vector<Tacka> tacke(n);
    for (int i = 0; i < n; i++) {
        int x, y;
        cin >> x >> y;
        tacke[i] = Tacka(x, y);
    }
    prostMnogougao(tacke);
    for (const Tacka& t : tacke)
        cout << t.x << " " << t.y << endl;
    return 0;
}

```

Уместо тежишта троугла одређеног са неке три неколинеарне тачке из скупа, за центар круга може се узети и једна од тачака из скупа — тачка z са највећом x -координатом (и са најмањом y -координатом, ако има више тачака са највећом x -координатом). Овакву тачку ћемо често користити приликом репавања геометријских проблема и зваћемо је *екстремна тачка*. Овако одабрану тачку повезујемо са свим осталим тачкама и тачке сортирамо растуће у односу на угао који полуправа од тачке z заклапа са x -осом. Ако две или више тачака заклапају исти угао са x -осом, оне се даље сортирају према растојању од тачке z и то на следећи начин: уколико првих неколико тачака заклапају исти угао, њих сортирамо у растућем редоследу растојања од тачке z , уколико је последњих неколико тачака колинеарно са тачком z њих сортирамо у опадајућем редоследу растојања од тачке z , док је за остале тачке које су колинеарне са тачком z све једно да ли ћемо их сортирати растуће или опадајуће.

```

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

```

```
using namespace std;
```

```

// tacka je zadata svojim dvema koordinatama
struct Tacka {

```

```

    int x, y;
    Tacka(int x_ = 0, int y_ = 0) {
        x = x_; y = y_;
    }
};

enum Orientacija { POZITIVNA, NEGATIVNA, KOLINEARNE };

Orientacija orijentacija(const Tacka& t0, const Tacka& t1, const Tacka& t2) {
    int d = (t1.x-t0.x)*(t2.y-t0.y) - (t2.x-t0.x)*(t1.y-t0.y);
    if (d > 0)
        return POZITIVNA;
    else if (d < 0)
        return NEGATIVNA;
    else
        return KOLINEARNE;
}

int kvadratRastojanja(const Tacka& t1, const Tacka& t2) {
    int dx = t1.x - t2.x, dy = t1.y - t2.y;
    return dx*dx + dy*dy;
}

void prostMnogougao(vector<Tacka>& tacke) {
    // trazimo tacku sa maksimalnom x koordinatom,
    // u slucaju da ima vise tacaka sa maksimalnom x koordinatom
    // biramo onu sa najmanjom y koordinatom
    auto max = max_element(begin(tacke), end(tacke),
        [](const Tacka& t1, const Tacka& t2) {
            return t1.x < t2.x ||
                (t1.x == t2.x && t1.y > t2.y);
        });
    // dovodimo je na pocetak niza - ona predstavlja centar kruga
    swap(*begin(tacke), *max);
    const Tacka& t0 = tacke[0];

    // sortiramo ostatak niza (tacke sortiramo na osnovu ugla koji
    // zaklapaju u odnosu vertikalnu polupravu koja polazi naviše iz
    // centra kruga), a kolinearne na osnovu rastojanja od centra kruga
    sort(next(begin(tacke)), end(tacke),
        [t0](const Tacka& t1, const Tacka& t2) {
            Orientacija o = orijentacija(t0, t1, t2);
            if (o == KOLINEARNE)
                return kvadratRastojanja(t0, t1) <= kvadratRastojanja(t0, t2);
            return o == POZITIVNA;
        });

    // obrcemo redosled tacaka na poslednjoj pravnoj
    auto it = prev(end(tacke));
    while (orijentacija(*prev(it), *it, t0) == KOLINEARNE)
        it = prev(it);
    reverse(it, end(tacke));
}

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

```

```

vector<Tacka> tacke(n);
for (int i = 0; i < n; i++) {
    int x, y;
    cin >> x >> y;
    tacke[i] = Tacka(x, y);
}
prostMnogougao(tacke);
for (const Tacka& t : tacke)
    cout << t.x << " " << t.y << endl;
return 0;
}

```

Задатак: Припадност тачке конвексном многоуглу

Напиши програм који утврђује да ли тачка припада конвексном многоуглу (ивице многоугла сматрају се његовим делом).

Улаз: Са стандардног улаза се учитава број темена конвексног многоугла n ($3 \leq n \leq 50000$), а затим редом темена у редоследу супротном од казаљке на сату. Свако теме се задаје у посебном реду, помоћу два цела броја раздвојена једним размаком. Након тога се задаје број m ($1 \leq m \leq 50000$) тачака чију припадност многоуглу треба испитати, а затим у наредних m редова координате тих тачака (координате су целобројне, раздвојене једним размаком).

Изаз: За сваку од m тачака на стандардни излаз у посебном реду исписати да ако тачка припада многоуглу тј. не у супротном.

Пример

Улаз	Изаз
6	da
4 0	da
2 2	ne
-2 2	ne
-4 0	da
-2 -2	
2 -2	
5	
-2 2	
-3 1	
-5 1	
3 -2	
0 0	

Решење

Сваки конвексни многоугао се може поделити на троуглове тако што се повуку све дијагонале из његовог произвољног темена. Тачка припада многоуглу ако и само ако припада неком од тих троуглова. Припадност тачке троуглу већ смо анализирали у задатку [Тачка у троуглу](#). Нагласимо да је проверу потребно мало прилагодити да би се у обзир узело то да се ивице многоугла сматрају његовим делом. Ако једна тачка лежи на датој дужи она ће се сматрати да је са исте стране те дужи као и било која друга тачка. Такође, пошто су вредности координата у овом задатку релативно велики бројеви, потребно је приликом израчунавања векторског производа обратити пажњу на могућност настанка прекорачења.

Пошто се припадност троуглу може испитати у времену $O(1)$, а посматрамо $n - 2$ троугла, сложеност испитивања припадности једне тачке је $O(n)$, док је сложеност провере за свих m тачака $O(mn)$.

```

#include <iostream>
#include <vector>
#include <utility>

```

```
using namespace std;
```

```
typedef pair<int, int> Tacka;
```



```

enum Orientacija {POZITIVNA, NEGATIVNA, KOLINEARNE};

Orientacija orijentacija(Tacka a, Tacka b, Tacka c) {
    int xa = a.first, ya = a.second;
    int xb = b.first, yb = b.second;
    int xc = c.first, yc = c.second;
    long long d = (long long)(xb-xa)*(long long)(yc-ya) -
        (long long)(xc-xa)*(long long)(yb-ya);

    if (d > 0)
        return POZITIVNA;
    else if (d < 0)
        return NEGATIVNA;
    else
        return KOLINEARNE;
}

bool saIsteStrane(const Tacka& T1, const Tacka& T2,
                 const Tacka& A1, const Tacka& A2) {
    Orientacija o1 = orijentacija(T1, T2, A1);
    Orientacija o2 = orijentacija(T1, T2, A2);
    if (o1 == KOLINEARNE || o2 == KOLINEARNE)
        return true;
    return o1 == o2;
}

bool tackaUTrouglu(const Tacka& T1, const Tacka& T2, const Tacka& T3,
                  const Tacka& A) {
    return saIsteStrane(T1, T2, T3, A) &&
        saIsteStrane(T1, T3, T2, A) &&
        saIsteStrane(T2, T3, T1, A);
}

bool sadrzi(const vector<Tacka>& poligon, const Tacka& A) {
    int n = poligon.size();
    for (int i = 1; i < n-1; i++)
        if (tackaUTrouglu(poligon[0], poligon[i], poligon[i+1], A))
            return true;
    return false;
}

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

    int n;
    cin >> n;
    vector<Tacka> poligon(n);
    for (int i = 0; i < n; i++) {
        cin >> poligon[i].first >> poligon[i].second;
    }

    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        Tacka A;
        cin >> A.first >> A.second;
        if (sadrzi(poligon, A))
            cout << "da" << '\n';
    }
}

```

```

    else
        cout << "ne" << '\n';
}

return 0;
}

```

Задатак можемо ефикасније решити бинарном претрагом. Свака дијагонала многоугао дели на два мања многоугао. Ако се установи да је тачка са једне стране дијагонале, знамо да она не може да припада оном од та два многоугла који лежи са супротне стране те дијагонале и потребно је испитати само да ли тачка припада оном другом многоуглу. Ако се дијагонала увек бира тако да та два многоугла имају приближно исти број темена, проблем ће се сводити на проблем истог облика, али двоструко мање димензије, што даје ефикасан алгоритам. Половљење прекидамо када остане троугао и тада проверу припадности троуглу вршимо слично као у задатку **Тачка у троуглу** (прилагодивши поступак томе да се ивице троугла сматрају његовим делом, као и могућности настанка прекорачења приликом израчунавања векторског производа, због релативно великих координата).

Пошто се у сваком кораку димензија проблема полови, сложеност припадности једне тачке је $O(\log n)$, док је сложеност провере за свих m тачака $O(m \log n)$.

```

#include <iostream>
#include <vector>
#include <utility>

using namespace std;

typedef pair<int, int> Tacka;

enum Orijentacija {POZITIVNA, NEGATIVNA, KOLINEARNE};

Orijentacija orijentacija(Tacka a, Tacka b, Tacka c) {
    int xa = a.first, ya = a.second;
    int xb = b.first, yb = b.second;
    int xc = c.first, yc = c.second;
    long long d = (long long)(xb-xa)*(long long)(yc-ya) -
                  (long long)(xc-xa)*(long long)(yb-ya);
    if (d > 0)
        return POZITIVNA;
    else if (d < 0)
        return NEGATIVNA;
    else
        return KOLINEARNE;
}

bool saIsteStrane(const Tacka& T1, const Tacka& T2,
                  const Tacka& A1, const Tacka& A2) {
    Orijentacija o1 = orijentacija(T1, T2, A1);
    Orijentacija o2 = orijentacija(T1, T2, A2);
    if (o1 == KOLINEARNE || o2 == KOLINEARNE)
        return true;
    return o1 == o2;
}

bool tackaUTrouglu(const Tacka& T1, const Tacka& T2, const Tacka& T3,
                   const Tacka& A) {
    return saIsteStrane(T1, T2, T3, A) &&
           saIsteStrane(T1, T3, T2, A) &&
           saIsteStrane(T2, T3, T1, A);
}

```

```

bool sadrzi(const vector<Tacka>& poligon, const Tacka& A, int l, int d) {
    if (d - l == 1)
        return tackaUTrouglu(poligon[0], poligon[l], poligon[d], A);
    int s = l + (d - l) / 2;
    if (orijentacija(poligon[0], poligon[s], A) == POZITIVNA)
        return sadrzi(poligon, A, s, d);
    else
        return sadrzi(poligon, A, l, s);
}

bool sadrzi(const vector<Tacka>& poligon, const Tacka& A) {
    int n = poligon.size();
    return sadrzi(poligon, A, 1, n-1);
}

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

    int n;
    cin >> n;
    vector<Tacka> poligon(n);
    for (int i = 0; i < n; i++) {
        cin >> poligon[i].first >> poligon[i].second;
    }

    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        Tacka A;
        cin >> A.first >> A.second;
        if (sadrzi(poligon, A))
            cout << "da" << '\n';
        else
            cout << "ne" << '\n';
    }

    return 0;
}

```

Задатак: Конвексни омотач

Потребно је оградити воћњак тако што ће се везати трака за нека стабла, тако да се након што трака обиђе пун круг сва стабла наћи унутар траке. Напиши програм који одређује за која стабла ће трака бити завезана.

Улаз: Са стандардног улаза се уноси број стабала n ($1 \leq n \leq 50000$), а затим у наредних n редова координате n стабала (два цела броја између -10^6 и 10^6 , раздвојена размаком).

Излаз: На стандардни излаз исписати број стабала за које ће бити везана трака, а затим, редом координате стабала за које се трака везује (ако су нека стабла колинеарна, траку је потребно везати само за она крајња од њих).

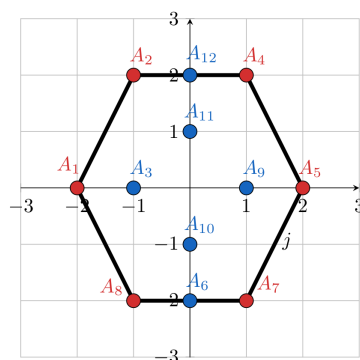
Пример

4.2. МНОГОУГЛОВИ

<i>Улаз</i>	<i>Изаз</i>
12	6
-2 0	2 0
-1 2	1 2
-1 0	-1 2
1 2	-2 0
2 0	-1 -2
0 -2	1 -2
1 -2	
-1 -2	
1 0	
0 -1	
0 1	
0 2	

Објашњење

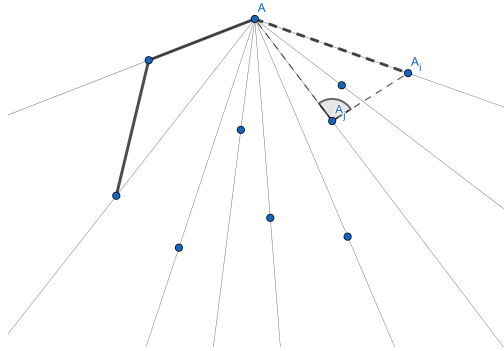
Положај стабала је приказан на слици. Трака се везује за стабла која су посебно означена.



Слика 4.4: Конвексни омотач

Решение

Задатак можемо решити “алгоритмом увијања поклона” (он се некада назива и Царвисов марш). Идеја је да изградњу конвексног омотача започнемо од неког екстремног темена (на пример, левог, доњег) и да у сваком кораку у омотач додамо наредну дуж која је део коначног конвексног омотача. Одређивање наредне дужи вршимо грубом силом. Њено једно теме је последња тачка до сада одређеног дела конвексног омотача. Да бисмо одредили друго теме, анализирамо све тачке и тражимо ону која даје дуж која са претходном дужи гради што већи угао (у полазном случају нема претходне дужи, па тражимо тачку која даје највећи угао са негативним делом осе u). Рецимо и да нема потребе за експлицитним рачунањем углова. Наиме, ако је претходно теме A , тада тачка A_i заклапа већи угао од тачке A_j ако и само ако је оријентација тројке $A_i A_j A$ позитивна. Потребно је још обратити пажњу на случај колинеарних тачака. Наиме, ако постоји више тачака које одређују максимални угао, тада се на тој полуправој узима она која је најдаља од претходног темена A . Поступак се завршава када се установи да је тачка која одређује највећи угао једнака почетној.



Слика 4.5: Увијање поклона

Овај алгоритам спада у групу алгоритама чија сложеност зависи од димензије излаза (а не само улаза). За додавање сваке нове излазне тачке обилазимо све улазне тачке па је сложеност $O(nm)$, где је n број улазних, а m број излазних тачака.

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

using namespace std;

// tacka je zadata svojim dvema koordinatama
struct Tacka {
    int x, y;
};

enum Orientacija { POZITIVNA, NEGATIVNA, KOLINEARNE };

Orientacija orijentacija(const Tacka& t0, const Tacka& t1, const Tacka& t2) {
    long long d = (long long)(t1.x-t0.x)*(long long)(t2.y-t0.y) -
                  (long long)(t2.x-t0.x)*(long long)(t1.y-t0.y);
    if (d > 0)
        return POZITIVNA;
    else if (d < 0)
        return NEGATIVNA;
    else
        return KOLINEARNE;
}

bool izmedju(const Tacka& t1, const Tacka& t2, const Tacka& t3) {
    return (t1.x < t2.x && t2.x < t3.x) ||
           (t1.x > t2.x && t2.x > t3.x) ||
           (t1.x == t2.x && t2.x == t3.x && t1.y < t2.y && t2.y < t3.y) ||
           (t1.x == t2.x && t2.x == t3.x && t1.y > t2.y && t2.y > t3.y);
}

vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
    vector<Tacka> omotac;
    auto cmp = [] (const Tacka& t1, const Tacka& t2) {
        return t1.x < t2.x || (t1.x == t2.x && t1.y < t2.y);
    };
    int prva = distance(begin(tacke), min_element(begin(tacke), end(tacke), cmp));
    int tekuca = prva;
    do {
```

```

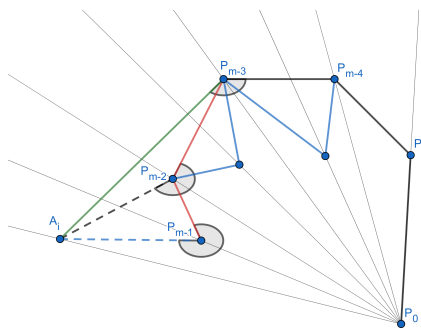
omotac.push_back(tacke[tekuca]);
int naredna = 0;
for (size_t i = 1; i < tacke.size(); i++) {
    Orijentacija o = orijentacija(tacke[tekuca], tacke[naredna], tacke[i]);
    if (naredna == tekuca ||
        o == POZITIVNA ||
        (o == KOLINEARNE && izmedju(tacke[tekuca], tacke[naredna], tacke[i])))
        naredna = i;
}
tekuca = naredna;
} while (tekuca != prva);
return omotac;
}

int main() {
    int n;
    cin >> n;
    vector<Tacka> tacke(n);
    for (int i = 0; i < n; i++)
        cin >> tacke[i].x >> tacke[i].y;
    vector<Tacka> omotac = konveksniOmotac(tacke);
    cout << omotac.size() << endl;
    for (const Tacka& t : omotac)
        cout << t.x << " " << t.y << endl;
    return 0;
}

```

Задатак ефикасно можемо решити Грахамовим алгоритмом. Прва фаза претпоставља конструкцију неког простог многоугла чија су темена дати скуп тачака. Нека су тачке тог многоугла редом $A_0 A_1 \dots A_{n-1}$. Ако се прост полигон конструише уобичајеним алгоритмом, његово прво теме A_0 ће бити крајње лево, доње теме и оно је сигурно екстремно. Обилазак вршимо у редоследу темена простог многоугла и претпостављамо да у сваком тренутку обилазка знамо и многоугао $P_0 P_1 \dots P_m$, који је део конвексни омотач до тог тренутка обрађених темена простог многоугла. Конвексни омотач полазне тачке A_0 је она сама, док је конвексни омотач прва два темена $\{A_0, A_1\}$ ивица $A_0 A_1$ која их спаја. Зато обраду прве две тачке вршимо просто тако што те две тачке додамо на почетак конвексног омотача тј. $P_0 = A_0$ и $P_1 = A_1$. Након тога, обрађујемо редом тачке A_i . Претпоставимо да је конвексни омотач тачака $A_0 A_1 \dots A_{i-1}$ полигон $P_0 P_1 \dots P_{m-1}$. Конвексни омотач скупа $\{A_0, A_1, \dots, A_i\}$ се састоји од неког почетног дела полигона $P_0 P_1 \dots P_m$ проширеног тачком A_i . Неке од завршних тачака полигона $P_0 P_1 \dots P_{m-1}$ неће више бити део конвексног омотача и оне се избацују. Тачка P_{m-1} није део конвексног омотача ако тројка $P_{m-2} P_{m-1} A_i$ није позитивно оријентисана. Ако се то установи, тада се тачка P_{m-1} уклања из конвексног омотача и исто се проверава за тачке, P_{m-2}, P_{m-3} и тако даље, све док се или не пронађе нека тачка код које је одговарајућа тројка позитивно оријентисана (тројка $P_0 P_1 A_i$ ће увек бити позитивно оријентисана).

На слици је приказана ситуација када се пре додавања тачке A_i из конвексног омотача уклањају две раније додате тачке.



Слика 4.6: Грахамов алгоритам

Сложеношћу алгоритма доминира изградња простог полигона (у њој се врши сортирање и та фаза је сложености $O(n \log n)$). Након тога, одређивање тачака које су део омотача је сложености $O(n)$ (иако постоји петља у петљи, свака тачка се обрађује једном и може највише једном бити додата и највише једном уклоњена из текућег конвексног омотача).

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

using namespace std;

// tacka je zadata svojim dvema koordinatama
struct Tacka {
    int x, y;
};

enum Orientacija { POZITIVNA, NEGATIVNA, KOLINEARNE };

Orientacija orientacija(const Tacka& t0, const Tacka& t1, const Tacka& t2) {
    long long d = (long long)(t1.x-t0.x)*(long long)(t2.y-t0.y) -
                 (long long)(t2.x-t0.x)*(long long)(t1.y-t0.y);
    if (d > 0)
        return POZITIVNA;
    else if (d < 0)
        return NEGATIVNA;
    else
        return KOLINEARNE;
}

long long kvadratRastojanja(const Tacka& t1, const Tacka& t2) {
    int dx = t1.x - t2.x, dy = t1.y - t2.y;
    return dx*dx + dy*dy;
}

void prostMnogougao(vector<Tacka>& tacke) {
    // trazimo tacku sa maksimalnom x koordinatom,
    // u slucaju da ima vise tacaka sa maksimalnom x koordinatom
    // biramo onu sa najmanjom y koordinatom
    auto max = max_element(begin(tacke), end(tacke),
        [](const Tacka& t1, const Tacka& t2) {
            return t1.x < t2.x ||
                (t1.x == t2.x && t1.y > t2.y);
        });
    // dovodimo je na pocetak niza - ona predstavlja centar kruga
```

```

swap(*begin(tacke), *max);
const Tacka& t0 = tacke[0];

// sortiramo ostatak niza (tačke sortiramo na osnovu ugla koji
// zaklapaju u odnosu vertikalnu polupravu koja polazi naviše iz
// centra kruga), a kolinearne na osnovu rastojanja od centra kruga
sort(next(begin(tacke)), end(tacke),
    [t0](const Tacka& t1, const Tacka& t2) {
        Orijentacija o = orijentacija(t0, t1, t2);
        if (o == KOLINEARNE)
            return kvadratRastojanja(t0, t1) <= kvadratRastojanja(t0, t2);
        return o == POZITIVNA;
    });

// obrcemo redosled tacaka na poslednjoj pravnoj
auto it = prev(end(tacke));
while (orijentacija(*prev(it), *it, t0) == KOLINEARNE)
    it = prev(it);
reverse(it, end(tacke));
}

vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
    vector<Tacka> omotac;
    prostMnogougao(tacke);
    omotac.push_back(tacke[0]);
    omotac.push_back(tacke[1]);
    for (size_t i = 2; i < tacke.size(); i++) {
        while (omotac.size() >= 2 &&
            orijentacija(omotac[omotac.size()-2],
                omotac[omotac.size()-1],
                tacke[i]) != POZITIVNA)
            omotac.pop_back();
        omotac.push_back(tacke[i]);
    }
    return omotac;
}

int main() {
    int n;
    cin >> n;
    vector<Tacka> tacke(n);
    for (int i = 0; i < n; i++)
        cin >> tacke[i].x >> tacke[i].y;
    vector<Tacka> omotac = konveksniOmotac(tacke);
    cout << omotac.size() << endl;
    for (const Tacka& t : omotac)
        cout << t.x << " " << t.y << endl;
    return 0;
}

```


Глава 5

Напредне структуре података

5.1 Префиксно дрво

Уређена бинарна дрвета омогућавају ефикасну имплементацију структура са асоцијативним приступом код којих се приступ врши по кључу који није целобројна вредност већ стринг или нешто друго. Још једна структура у виду дрвета која омогућава ефикасан асоцијативан приступ је *префиксно дрво* такође познато под енглеским називом *trie* (од енглеске речи *reTRIEval*). Основна идеја ове структуре је да путање од корена до листова или до неких унутрашњих чворова кодирају кључеве, а да се подаци везани за тај кључ чувају у чвору док којег се долази проналажењем кључа дуж путање. У случају ниски, корен садржи празну реч, а преласком преко сваке гране се дна до тада формирану реч надовезује још један карактер. Притом, заједнички префиксни различитих кључева су представљени истим путањама од корена до тачке разликовања. Један пример оваког дрвета, код којег су приказане ознаке придружене гранама, а не чворовима, је следећи:

```
      a      n      d
     n  t   o  a  u
    a      ć ž  n  h ž
```

Кључеви које дрво чува су *ana*, *at*, *noć*, *nož*, *da*, *dan*, *duh* и *duž*. Приметимо да се кључ *da* не завршава листом и да стога сваки чвор мора чувати информацију о томе да ли се њиме комплетира неки кључ (и у том случају садржати податак) или не. Илустрације ради, могу се приказати ознаке на чворовима, које представљају префиксе акумулиране до тих чворова. Треба имати у виду да овај приказ не илуструје имплементацију, већ само префиксе дуж гране. Симбол @ представља празну реч.

```
              @
             n      d
          a      n      d
        an  at   no  da  du
       ana   noć nož dan duh duž
```

У случају да неки чвор има само једног потомка и не представља крај неког кључа, грана до њега и грана од њега се могу спојити у једну, њихови карактери надовезати, а чвор елиминисати. Овако се добија компактнија репрезентација префиксног дрвета.

Поред општег асоцијативног приступа подацима, очигледна примена ове структуре је и имплементација коначних речника, на пример у сврхе аутоматског комплетирања или провере исправности речи које корисник куча на рачунару или мобилном телефону.

Напоменимо још и да ова структура није резервисана за чување стрингова. На пример, у случају целих бројева или бројева у покретном зарезу, кључ могу бити ниске битова које представљају такве бројеве.

У случају коначне азбуке величине m , сложеност операција у најгорем случају је $O(mn)$, где је n дужина речи која се тражи, умеће или брише, док је сложеност ових операција у просечном случају $O(n)$. Када би се уместо префиксног дрвета користило балансирано уређено бинарно дрво, сложеност ових операција би у најгорем случају била $O(M \log N)$, где је са N означен укупан број кључева који се чувају у дрвету, а са M максимална дужина кључа. Уколико су кључеви релативно кратки, предност префиксног дрвета је што сложеност зависи од дужине записа кључа, а не од броја елемената у дрвету. Мана је потреба за чувањем

5.1. ПРЕФИКСНО ДРВО

показивача уз сваки карактер у дрвету. Штавише, просторна сложеност префиксног дрвета је $O(M \cdot m \cdot N)$, где је са N означен број кључева који се чувају у префиксном дрвету, а са M максимална дужина кључа.

Претрага и уметање се праволинијски имплементирају, док је у случају брисања некада потребно брисати више од једног чвора.

Задатак: Днк секвенце

ДНК секвенце се представљају нискама које се састоје од карактера `c`, `t`, `a` и `g`. У програму се одржава скуп подсеквенци једне дате генетске секвенце. Програм подржава три операције: убацивање подсеквенце у скуп (ако је подсеквенца већ у скупу, ова операција нема ефекта), избацивање подсеквенце из скупа (ако се подсеквенца не налази у скупу, ова операција нема ефекта) и испитивање да ли се подсеквенца налази у скупу.

Улаз: Са стандардног улаза се уноси број упита n ($1 \leq n \leq 50000$), а затим у наредних n редова по један упит. Упит може бити облика `ubaci niska`, `izbaci niska` или `trazi niska`, при чему се свака ниска састоји само од карактера `c`, `t`, `a` и `g` и дугачка је између једног и 5000 карактера.

Издаз: За сваки упит `trazi` исписати `da` ако се тражена ниска тренутно налази у скупу или `ne` ако се не налази.

Пример

Улаз	Издаз
14	da
ubaci cta	da
ubaci ct	da
ubaci ctg	ne
trazi cta	ne
trazi ct	da
trazi ctg	ne
trazi ctc	da
izbaci ct	
trazi ct	
trazi ctg	
izbaci ctg	
trazi ctg	
ubaci ctg	
trazi ctg	

Решење

У програму се тражи одржавање скупа ниски. Најједноставније, а и најефикасније решење је да користимо библиотечку имплементацију скупа.

```
#include <iostream>
#include <unordered_set>

using namespace std;

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

    unordered_set<string> niske;
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        string naredba, niska;
        cin >> naredba >> niska >> ws;
        if (naredba == "ubaci")
            niske.insert(niska);
        else if (naredba == "izbaci")
            niske.erase(niska);
    }
```

```

    else if (naredba == "trazi") {
        if (niske.find(niska) != niske.end())
            cout << "da" << "\n";
        else
            cout << "ne" << "\n";
    }
}
return 0;
}

```

У сваком чвору префиксног дрвета можемо чувати асоцијативни низ који пресликава слова азбуке у наредне чворове дрвета. Функције можемо имплементирати рекурзивно.

```

#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
using namespace std;

// структура чвора prefiksnog drveta - u svakom čvoru čuvamo niz
// grana obeleženih karakterima ka potomcima i informaciju da li
// je u ovom čvoru kraj neke reči
struct Cvor {
    bool krajReci = false;
    unordered_map<char, Cvor*> grane;
};

// tražimo sufiks reči w koji počinje od pozicije i u drvetu na
// čiji koren ukazuje pokazivač drvo
bool sadrzi(Cvor *drvo, const string& w, size_t i) {
    // ako je drvo prazno, ono ne sadrži traženu reč
    if (drvo == nullptr)
        return false;

    // ako je sufiks prazan, on je u korenu akko je u korenu obeleženo
    // da je tu kraj reči
    if (i == w.size())
        return drvo->krajReci;

    // tražimo granu na kojoj piše w[i]
    auto it = drvo->grane.find(w[i]);
    // ako je nađemo, rekurzivno tražimo ostatak sufiksa od pozicije i+1
    if (it != drvo->grane.end())
        return sadrzi(it->second, w, i+1);

    // nismo našli granu sa w[i], pa reč ne postoji
    return false;
}

// tražimo reč w u drvetu na čiji koren ukazuje pokazivač drvo
bool sadrzi(Cvor* drvo, const string& w) {
    return sadrzi(drvo, w, 0);
}

// umeće sufiks reči w od pozicije i u drvo na čiji koren ukazuje
// pokazivač drvo
void ubaci(Cvor* drvo, const string& w, size_t i) {
    // ako je sufiks prazan samo u korenu beležimo da je tu kraj reči

```

```
if (i == w.size()) {
    drvo->krajReci = true;
    return;
}

// tražimo granu na kojoj piše w[i]
auto it = drvo->grane.find(w[i]);
// ako takva grana ne postoji, dodajemo je kreirajući novi čvor
if (it == drvo->grane.end())
    drvo->grane[w[i]] = new Cvor();

// sada znamo da grana sa w[i] sigurno postoji i preko te grane
// nastavljamo dodavanje sufiksa koji počinje na i+1
ubaci(drvo->grane[w[i]], w, i+1);
}

// umeće reč w u drvo na čiji koren ukazuje pokazivač drvo
Cvor* ubaci(Cvor *drvo, string& w) {
    if (drvo == nullptr)
        drvo = new Cvor();
    ubaci(drvo, w, 0);
    return drvo;
}

// izbacuje iz drвета na čiji koren ukazuje pokazivač drvo sufiks reči
// w od pozicije
// funkcija vraća informaciju o tome da li nakon brisanja ostaje prazno drvo
bool izbac(Cvor* drvo, const string& w, size_t i) {
    // ako je drvo prazno, ništa ne treba menjati
    if (drvo == nullptr)
        // drvo je bilo i ostalo prazno
        return true;

    // ako smo stigli do kraja reči
    if (i == w.length())
        // označavamo da u tekućem čvoru nije više kraj neke reči
        drvo->krajReci = false;
    else {
        auto it = drvo->grane.find(w[i]);
        // u suprotom brišemo sufiks i ako se nakon toga dobije prazno
        // drvo, uklanjamo granu
        if (it != drvo->grane.end() && izbac(it->second, w, i+1))
            drvo->grane.erase(it);
    }

    // ako je u drvetu ostao jedan potpuno prazan čvor
    if (!drvo->krajReci && drvo->grane.size() == 0) {
        // brišemo ga
        delete drvo;
        // javljamo da je nakon brisanja nastalo prazno drvo
        return true;
    } else
        // javljamo da drvo nakon brisanja nije prazno
        return false;
}

// iz drвета na koji ukazuje pokazivač drvo brišemo reč w
Cvor* izbac(Cvor* drvo, const string& w) {
```

```

    if (izbaci(drvo, w, 0))
        return nullptr;
    else
        return drvo;
}

// funkcija brise prefiksno drvo sa korenom koren
void obrisi(Cvor* koren) {
    if (koren != nullptr) {
        for (auto it : koren->grane)
            obrisi(it.second);
        delete koren;
    }
}

// program kojim testiramo gornje funkcije
int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);

    Cvor* koren = nullptr;
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        string naredba, niska;
        cin >> naredba >> niska >> ws;
        if (naredba == "ubaci")
            koren = ubaci(koren, niska);
        else if (naredba == "izbaci")
            koren = izbaci(koren, niska);
        else if (naredba == "trazi") {
            if (sadrzi(koren, niska))
                cout << "da" << "\n";
            else
                cout << "ne" << "\n";
        }
    }
    obrisi(koren);
    return 0;
}

```

У случају када је азбука мала (што је случај у овом задатку), пресликавање карактера у наредне чворове дрвета можемо чувати у низу (на местима карактера који немају наследнике чуваћемо null показиваче). Пошто је за брисање битно да знамо колико наследника има неки чвор, у чвору ћемо чувати и ту информацију. Функције за убацивање и претрагу дрвета једноставно можемо имплементирати и итеративно. Са друге стране, избацивање није једноставно имплементирати итеративно (јер је након брисања речи потребно вратити се уз дрво и обрисати све празне чворове, а немамо показиваче ка родитељским чворовима којима бисмо се једноставно могли враћати унатраг).

```

#include <iostream>
using namespace std;

// karaktere iz azbuke kodiramo brojevima
int kod(char c) {
    switch(c) {
        case 'c': return 0;
        case 't': return 1;
        case 'g': return 2;
        case 'a': return 3;
    }
}

```

```

    return -1;
}

// cvor prefiksnog drveta
struct Cvor
{
    Cvor* grane[4];
    int brojGrana;
    bool krajReci;
};

// kreira se novi, prazan cvor
Cvor* noviCvor() {
    Cvor* novi = new Cvor();
    fill(novi->grane, novi->grane+4, nullptr);
    novi->brojGrana = 0;
    novi->krajReci = false;
    return novi;
}

// ubacivanje reci str u prefiksno drvo sa korenom koren
// funkcija vraca novi koren drveta
Cvor* ubaci(Cvor* koren, const string& str) {
    if (koren == nullptr)
        koren = noviCvor();

    Cvor* cvor = koren;
    for (char c : str) {
        if (cvor->grane[kod(c)] == nullptr) {
            cvor->grane[kod(c)] = noviCvor();
            cvor->brojGrana++;
        }
        cvor = cvor->grane[kod(c)];
    }
    cvor->krajReci = true;

    return koren;
}

// proverava da li prefiksno drvo sa korenom koren sadrzi rec str
bool sadrzi(Cvor* koren, const string& str) {
    if (koren == nullptr)
        return false;
    Cvor* cvor = koren;
    for (char c : str) {
        if (cvor->grane[kod(c)] == nullptr)
            return false;
        cvor = cvor->grane[kod(c)];
    }
    return cvor != nullptr && cvor->krajReci;
}

// funkcija izbacuje sufiks reci str od pozicije i iz drveta sa korenom koren
// funkcija vraca koren drveta nastalog nakon izbacivanja
Cvor* izbaci(Cvor* koren, const string& str, size_t i) {
    // ako je drvo vec prazno, nema sta da se izbacuje
    if (koren == nullptr)

```

```

    return koren;

    // ako smo dosli do kraja reci
    if (i == str.length())
        // rec brisemo tako sto oznacavamo da u tom cvoru vise nije kraj neke reci
        koren->krajReci = false;
    else {
        // ako postoji grana sa sa tekucim slovom reci
        if (koren->grane[kod(str[i])] != nullptr) {
            // brisemo ostatak te reci iz drveta
            Cvor* c = izbaci(koren->grane[kod(str[i])], str, i+1);
            // ako se tim brisanjem potpuno uklonio deo drveta ispod te grane
            // onda uklanjamo i tu granu
            if (c == nullptr)
                koren->brojGrana--;
            koren->grane[kod(str[i])] = c;
        }
    }

    // ako je nakon brisanja koren potpuno prazan
    if (!koren->krajReci && koren->brojGrana == 0) {
        // brisemo ga i vracamo null kao oznaku praznog stabla
        delete koren;
        return nullptr;
    } else
        // vracamo tekuci koren
        return koren;
}

// funkcija izbacuje rec str iz drveta sa korenom koren
// funkcija vraci koren drveta nastalog nakon izbacivanja
Cvor* izbaci(Cvor* cvor, const string& str) {
    return izbaci(cvor, str, 0);
}

// funkcija brise prefiksno drvo sa korenom koren
void obrisi(Cvor* koren) {
    if (koren != nullptr) {
        for (Cvor* c : koren->grane)
            obrisi(c);
        delete koren;
    }
}

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

    Cvor* koren = nullptr;
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        string naredba, niska;
        cin >> naredba >> niska >> ws;
        if (naredba == "ubaci")
            koren = ubaci(koren, niska);
        else if (naredba == "izbaci")
            koren = izbaci(koren, niska);
    }
}

```

```

    else if (naredba == "trazi") {
        if (sadrzi(koren, niska))
            cout << "da" << "\n";
        else
            cout << "ne" << "\n";
    }
}
obrisi(koren);
return 0;
}

```

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

Једно могуће решење задатка је да се бројеви убаце у префиксно дрво. Приликом убацивања сваког новог броја проверавамо да ли је неки број у дрвету његов префикс и да ли је он префикс неког броја у дрвету. Једна могућа имплементација је она у којој се у сваком чвору дрвета чува асоцијативни низ (мапа тј. речник) који пресликава цифре у нове чворове дрвета.

```

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

struct Cvor {
    unordered_map<char, Cvor*> grane;
};

Cvor* napravi(const string& s, size_t i) {
    if (i == s.size())
        return nullptr;
    Cvor* c = new Cvor();
    c->grane[s[i]] = napravi(s, i+1);
    return c;
}

bool ubaci(Cvor* cvor, const string& s, size_t i) {
    if (i == s.size() || cvor == nullptr)
        return false;
    auto it = cvor->grane.find(s[i]);
    if (it == cvor->grane.end()) {
        cvor->grane[s[i]] = napravi(s, i+1);
        return true;
    } else
        return ubaci(it->second, s, i+1);
}

```



```

void obrisi(Cvor* cvor) {
    if (cvor != nullptr) {
        for (auto it : cvor->grane)
            obrisi(it.second);
        delete cvor;
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    int n;
    cin >> n;
    bool OK = true;
    string broj;
    cin >> broj;
    Cvor* koren = napravi(broj, 0);
    for (int i = 1; i < n; i++) {
        cin >> broj;
        if (OK && !ubaci(koren, broj, 0))
            OK = false;
    }
    if (OK)
        cout << "da" << endl;
    else
        cout << "ne" << endl;
    obrisi(koren);

    return 0;
}

```

Још једна могућа имплементација префиксног дрвета је она у којој се у сваком чвору чува низ показивача ка наследницима, који има онолико елемената колика је величина азбуке (у овом задатку азбуку чине цифре, па чувамо 10 елементне низове). У сваком чвору тада морамо чувати и информацију о томе да ли се нека реч завршава у том чвору.

```

#include <iostream>
using namespace std;

struct Cvor {
    Cvor* grane[10];
    bool krajReci;
};

Cvor* noviCvor() {
    Cvor* novi = new Cvor();
    fill(novi->grane, novi->grane+10, nullptr);
    novi->krajReci = false;
    return novi;
}

bool ubaci(Cvor* koren, const string& s) {
    Cvor* cvor = koren;
    bool OK = false;
    for (char c : s) {
        if (cvor->krajReci)
            return false;
        if (cvor->grane[c - '0'] == nullptr) {
            OK = true;

```

```

        cvor->grane[c - '0'] = noviCvor();
    }
    cvor = cvor->grane[c - '0'];
}
if (!OK)
    return false;
cvor->krajReci = true;
return true;
}

void obrisi(Cvor* cvor) {
    if (cvor != nullptr) {
        for (auto it : cvor->grane)
            obrisi(it);
        delete cvor;
    }
}

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

    int n;
    cin >> n;
    Cvor* koren = noviCvor();
    bool OK = true;
    for (int i = 0; i < n; i++) {
        string broj;
        cin >> broj;
        if (OK && !ubaci(koren, broj))
            OK = false;
    }
    if (OK)
        cout << "da" << endl;
    else
        cout << "ne" << endl;
    obrisi(koren);

    return 0;
}

```

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;
    }
}

```

Задатак: Днк префикси

Напиши програм који омогућава унос генетских секвенци (ниски које се састоје само од карактера а, с, т и г) и који за сваку унету секвенцу исписује колико има секвенци којима је секвенца која се уноси префикс.

Улаз: Свака линија стандардног улаза садржи ниску која садржи највише 10000 карактера а, с, т или г.

Израз: За сваку унету ниску исписати колико има раније унетих ниски којима је она префикс.

Пример

Улаз	Израз
tacg	0
tac	1
ta	2
ca	0
cat	0

Решење

Задатак можемо решити коришћењем префиксног дрвета тако што ћемо у сваком чвору дрвета чувати број ниски којима је ниска представљена тим чвором префикс. Током убацивања ниске увећаваћемо бројач свих чворова кроз које се пролази током убацивања (то су сви њени префикси).

```

#include <iostream>

using namespace std;

// karaktere iz azbuke kodiramo brojevima
int kod(char c) {
    switch(c) {
        case 'c': return 0;
        case 't': return 1;
        case 'g': return 2;
        case 'a': return 3;
    }
    return -1;
}

// cvor prefiksnog drveta
struct Cvor
{
    Cvor* grane[4];
    int brojPrefiksa;
};

// kreira se novi, prazan cvor
Cvor* noviCvor() {
    Cvor* novi = new Cvor();
    fill(novi->grane, novi->grane+4, nullptr);
    novi->brojPrefiksa = 0;
}

```

```

    return novi;
}

// ubacivanje reci str u prefiksno drvo sa korenom koren
// funkcija vraca novi koren drveta
void ubaci(Cvor* koren, const string& str) {
    Cvor* cvor = koren;
    for (char c : str) {
        if (cvor->grane[kod(c)] == nullptr)
            cvor->grane[kod(c)] = noviCvor();
        cvor = cvor->grane[kod(c)];
        cvor->brojPrefiksa++;
    }
}

// ocitavanje broja reci u drvetu sa korenom koren koje za prefiks
// imaju rec str
int brojPrefiksa(Cvor* koren, const string& str) {
    Cvor* cvor = koren;
    for (char c : str) {
        if (cvor->grane[kod(c)] == nullptr)
            return 0;
        cvor = cvor->grane[kod(c)];
    }
    if (cvor == nullptr)
        return 0;
    return cvor->brojPrefiksa;
}

// brise drvo sa korenom koren
void obrisi(Cvor* koren) {
    if (koren != nullptr) {
        for (Cvor* c : koren->grane)
            obrisi(c);
        delete koren;
    }
}

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);
    // prefiksno drvo u kome se cuvajuci ucitane niske i broje prefiksi
    Cvor* drvo = noviCvor();
    // ucitavamo jednu po jednu nisku do kraja ulaza
    string niska;
    while (cin >> niska) {
        // ispisujemo broj niski u drvetu kojima je tekuca niska prefiks
        cout << brojPrefiksa(drvo, niska) << '\n';
        // ubacujemo novu nisku u drvo
        ubaci(drvo, niska);
    }
    // brisemo prefiksno drvo
    obrisi(drvo);
    return 0;
}

```

Joш једна могућност је да се учитане ниске смештају у мултикуп (сортиран лексикографски). Број niskи које за префикс имају дату ниску s се може ефикасно одредити тако што се пронађе прва ниска у мултискупу која је лексикографски већа или једнака s и затим се итерира редом кроз ниске све док им је ниска s префикс.

```

#include <iostream>
#include <set>

using namespace std;

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);
    multiset<string> niske;
    string niska;
    while (cin >> niska) {
        auto it = niske.lower_bound(niska);
        int broj = 0;
        while (it != niske.end() && it->compare(0, niska.size(), niska) == 0) {
            broj++;
            it++;
        }
        cout << broj << '\n';
        niske.insert(niska);
    }
    return 0;
}

```

Задатак: Различите подниске

Напиши програм који одређује колико различитих подниски има дата ниска.

Улаз: Са стандардног улаза се уноси ниска дужине највише 5000 карактера, која се састоји само од малих слова a, b, c и d.

Излаз: На стандардни излаз исписати тражени број различитих подниски

Пример

Улаз	Излаз
ababa	10

Објашњење

Различите подниске су "", "a", "ab", "aba", "abab", "ababa", "b", "ba", "bab" и "baba".

Решење

Задатак можемо решити тако што одржавамо скуп који ће садржати све подниске учитане ниске. Све подниске лако можемо набројати коришћењем угнежђених петљи. Број подниски је квадратни тј. $O(n)$. Уметање у скуп који садржи n елемената може реализовати у сложености $O(\log n)$ ако се користи балансирано дрво, међутим, не треба занемарити да се приликом уметања пореде дугачке ниске, што може бити скупа операција (сложености $O(n)$), па уметање ниске може захтевати $O(n \log n)$ корака. Укупна временска сложеност је $O(n^3 \log n)$. Ако је скуп реализован у облику хеш-мапе, тада можемо очекивати да ће унос у просеку захтевати константно време, али је време потребно за израчунавање хеш вредности дугачких ниски незанемарив фактор и линеарно зависи од дужине ниске, па се и у овом случају може очекивати укупна сложеност $O(n^3)$. Већи проблем је заузеће меморије које је $O(n^3)$.

```

#include <iostream>
#include <set>

using namespace std;

int main() {
    string s;
    cin >> s;
    set<string> podniske;
    podniske.insert("");
    for (size_t i = 0; i < s.length(); i++)

```

```

    for (size_t duzina = 1; duzina <= s.length() - i; duzina++)
        podniske.insert(s.substr(i, duzina));
    cout << podniske.size() << endl;
    return 0;
}

```

Сваки чвор у префиксном дрвету представља префикс неке од убачених ниски, при чему се заједнички префикси деле између свих убачених ниски. Зато је укупан број чворова у дрвету једнак укупном броју различитих префикса свих убачених подниски. Свака подниска неке ниске је префикс неког њеног суфикса. Стога задатак можемо решити тако што у префиксно дрво убацимо све суфиксе учитане ниске, а затим пребројимо чворове дрвета.

Уметање ниске дужине m у дрво је сложености $O(m)$. Дужине суфикса су редом од 1 до n , па је укупно време потребно за формирање дрвета $O(n^2)$. Одређивање броја чворова дрвета је линеарна у односу на тај број чворова, па пошто њих може бити $O(n^2)$, укупна и временска и меморијска сложеност овог приступа је $O(n^2)$.

```

#include <iostream>
#include <algorithm>

using namespace std;

const int BROJ_SLOVA_AZBUKE = 4;

struct Cvor {
    Cvor* grane[BROJ_SLOVA_AZBUKE];
};

Cvor* noviCvor() {
    Cvor* novi = new Cvor();
    fill(novi->grane, novi->grane + BROJ_SLOVA_AZBUKE, nullptr);
    return novi;
}

void ubaci(Cvor* koren, const string& str, size_t i) {
    Cvor* cvor = koren;
    while (i < str.length()) {
        if (cvor->grane[str[i] - 'a'] == nullptr)
            cvor->grane[str[i] - 'a'] = noviCvor();
        cvor = cvor->grane[str[i] - 'a'];
        i++;
    }
}

int brojCvorova(Cvor* koren) {
    if (koren == nullptr)
        return 0;

    int broj = 1;
    for (int i = 0; i < BROJ_SLOVA_AZBUKE; i++)
        broj += brojCvorova(koren->grane[i]);

    return broj;
}

void obrisi(Cvor* koren) {
    if (koren != nullptr) {
        for (int i = 0; i < BROJ_SLOVA_AZBUKE; i++)
            obrisi(koren->grane[i]);
        delete koren;
    }
}

```

```

    }
}

int main() {
    string s;
    cin >> s;
    Cvor* koren = noviCvor();
    for (size_t i = 0; i < s.length(); i++)
        ubaci(koren, s, i);
    cout << brojCvorova(koren) << endl;
    obrisi(koren);
    return 0;
}

```

Задатак: Пар који даје највећи XOR

Напиши програм који међу унетим природним бројевима одређује онај пар који даје највећи резултат при операцији ексклузивне дисјункције (xor).

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 100000$), а затим n природних бројева између 0 и 10^{18} , сваки у посебном реду.

Излаз: На стандардни излаз исписати максималну вредност која се може добити када се операција ексклузивне дисјункције примени на нека два унета броја.

Пример

Улаз	Излаз
5	7
1	
2	
3	
4	
5	

Објашњење

Највећи резултат 7 добија се ексклузивном дисјункцијом бројева 3 и 4 (њихови бинарни записи су 00...0000011 и 000...000100). Исти резултат добија се и ексклузивном дисјункцијом бројева 2 и 5 (њихови бинарни записи су 0000...0000101 и 0000...00000010).

Решење

Решење грубом силом подразумева да се на сваки пар бројева примени операција XOR и да се испише максимум добијених резултата. Сложеност овог приступа је $O(n^2)$.

```

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

typedef unsigned long long ull;

int main() {
    int n;
    cin >> n;
    vector<ull> brojevi(n);
    for (int i = 0; i < n; i++)
        cin >> brojevi[i];
    ull max = 0;
    for (int i = 1; i < n; i++)
        for (int j = 0; j < i; j++)
            if ((brojevi[i] ^ brojevi[j]) > max)
                max = brojevi[i] ^ brojevi[j];
}

```

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

Ефикасније решење можемо добити применом структура података. Покушајмо да за сваки нови унети број ефикасно израчунамо највећи број који се може добити применом операције XOR на њега и неки од претходно унетих бројева (у решењу грубом силом, то се дешава у унутрашњој петљи). Покушајмо да тај број одредимо бит-по-бит и то кренувши од битова највеће тежине. На месту бита највеће тежине можемо добити 1 ако текући број почиње битом 0 и међу раније учитаним бројевима постоји неки који почиње битом 1 или ако текући број почиње битом 1 и међу раније учитаним бројевима постоји неки који почиње битом 0. У супротном на месту највеће тежине резултата мора бити бит 0. Након одређивања првог бита, одређујемо наредни, али у случају да смо на водеће место резултата уписали 1, међу учитаним нискама задржавамо само оне које су на водећем месту имали бит супротан бит водећем биту текућег броја (у случају да смо на водеће место резултата уписали 0, тада су сви раније учитани бројеви почињали истим битом којим почиње текући број и сви се задржавају). Поступак сада понављамо за други бит, при чему разматрамо само ниске које нису раније одбачене. На пример, ако су дати бројеви

```
1001
1010
0110
```

и ако је текући број

```
0010
```

На водеће место резултата можемо уписати 1, при чему задржавамо ниске

```
1001
1010
```

На друго место резултата морамо уписати 0, јер све задржане ниске на месту другог бита имају 0, исто као и текући број. Обе ниске се задржавају.

На треће место резултата можемо уписати 1 и тада задржавамо само ниску

```
1001
```

На крају, на последње место резултата можемо уписати 1. Коначан резултат је, дакле, 1011 и он се добија применом операције XOR на ниске 0010 и 1001.

Претрагу можемо веома једноставно организовати ако све учитане бинарне записе упишемо у префиксно дрво. У сваком нивоу дрвета одлучујемо се за један бит и спуштањем на наредни ниво елиминишемо све оне ниске које на одговарајућем месту немају тај бит. Ако у текућем кораку постоји грана обележена битом супротном од текућег бита тренутног броја, спуштамо се њоме и у резултат уписујемо јединицу, док се у супротном спуштамо граном на којој се налази бит једнак текућем биту тренутног броја и у резултат уписујемо нулу.

Сложеност овог алгоритма је $O(n)$, при чему је константа једнака броју битова којима се записују бројеви (с обзиром на ограничења дата у задатку, то је 64).

```
#include <iostream>
```

```
using namespace std;
```

```
typedef unsigned long long ull;
```

```
struct Cvor {
    Cvor* grane[2];
};
```

```
Cvor* noviCvor() {
    Cvor* novi = new Cvor();
    novi->grane[0] = novi->grane[1] = nullptr;
    return novi;
}
```



```

void ubaci(Cvor* koren, ull broj) {
    Cvor* cvor = koren;
    ull mask = 1ull << (8*sizeof(ull) - 1);
    while (mask != 0) {
        int bit = (broj & mask) != 0;
        if (cvor->grane[bit] == nullptr)
            cvor->grane[bit] = noviCvor();
        cvor = cvor->grane[bit];
        mask >>= 1;
    }
}

ull maxXOR(Cvor* koren, ull broj) {
    Cvor* cvor = koren;
    ull rez = 0;
    ull mask = 1ull << (8*sizeof(ull) - 1);
    while (mask != 0) {
        int bit = (broj & mask) != 0;
        if (cvor->grane[!bit] != nullptr) {
            rez = rez | mask;
            cvor = cvor->grane[!bit];
        } else
            cvor = cvor->grane[bit];
        mask >>= 1;
    }
    return rez;
}

void obrisi(Cvor* koren) {
    if (koren != nullptr) {
        obrisi(koren->grane[0]);
        obrisi(koren->grane[1]);
        delete koren;
    }
}

int main() {
    int n;
    cin >> n;
    unsigned long long x;
    cin >> x;
    Cvor* koren = noviCvor();
    ubaci(koren, x);
    unsigned long long max = 0;
    for (int i = 1; i < n; i++) {
        cin >> x;
        unsigned long long rez = maxXOR(koren, x);
        if (rez > max)
            max = rez;
        ubaci(koren, x);
    }
    cout << max << endl;
    obrisi(koren);
    return 0;
}

```

Задатак: Највећи XOR сегмента

Напиши програм који у низу природних бројева одређује највећи број који се може добити операцијом ексклузивне дисјункције примењене на неки непразни сегмент (подниз узастопних елемената) тог низа.

Улаз: Са стандардног улаза се учитава број елемената низа n ($1 \leq n \leq 100000$), а затим из n наредних редова по један природни број између 1 и 10^{18} .

Излаз: На стандардни излаз исписати тражену максималну вредност ексклузивне дисјункције неког непразног сегмента низа.

Пример

Улаз	Излаз
5	54
10	
20	
30	
40	
50	

Објашњење

Максимална вредност се постиже за двочлани сегмент 30 40 (бинарни записи ових бројева су ...011110 и ...101000, а резултата је ...110110).

Решење

Решење грубом силом подразумева да се за сваки сегмент израчуна вредност ексклузивне дисјункције. Сегменте можемо набрајати угнежђеним петљама где се у спољној петљи мења леви, а у унутрашњој петљи десни крај сегмента. На тај начин се сегменти проширују једним по једним елементом, па вредност ексклузивне дисјункције њихових елемената можемо израчунавати инкрементално.

Сложеност оваквог алгорита је $O(n^2)$.

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

```
using namespace std;
```

```
typedef unsigned long long ull;
```

```
int main() {
    int n;
    cin >> n;
    ull x;
    cin >> x;
    ull xorPrefiksa = x;
    vector<ull> xoroviPrefiksa(n);
    xoroviPrefiksa[0] = xorPrefiksa;
    ull maksSegmenta = xorPrefiksa;
    for (int i = 1; i < n; i++) {
        cin >> x;
        xorPrefiksa = xorPrefiksa ^ x;
        maksSegmenta = max(maksSegmenta, xorPrefiksa);
        for (int j = 0; j < i; j++)
            maksSegmenta = max(maksSegmenta, xorPrefiksa ^ xoroviPrefiksa[j]);
        xoroviPrefiksa[i] = xorPrefiksa;
    }
    cout << maksSegmenta << endl;
    return 0;
}
```

Задатак можемо ефикасније решити коришћењем технике сличне техници изражавања збира сегмента као разлике два збира префикса. Наиме, важи да је

$$\bigoplus_{k=i}^j a_k = \bigoplus_{k=0}^j a_k \oplus \bigoplus_{k=0}^{i-1} a_k$$

Зато у програму можемо одржавати низ свих досадашњих вредности ексклузивних дисјункција префикса низа. Приликом анализе наредног елемента одређујемо ексклузивну дисјункцију префикса који се њиме завршава (то можемо урадити инкрементално, јер већ знамо ексклузивну дисјункцију елемената претходног префикса). Поредимо ту вредност са тренутним максимумом, а затим из њега изузимамо један по један елемент са почетка и упоређујемо тако добијене вредности са тренутним максимумом (наравно, ажурирамо максимум кад год је то потребно). На крају вредност тренутног префикса додајемо у низ вредности претходних префикса и прелазимо на наредни елемент, све док не обрадимо све елементе низа.

Пошто анализирамо n елемената низа и пошто приликом анализе сваког елемента анализирамо све претходне префиксе, сложеност овог приступа је $O(n^2)$.

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

using namespace std;

typedef unsigned long long ull;

int main() {
    int n;
    cin >> n;
    vector<ull> brojevi(n);
    for (int i = 0; i < n; i++)
        cin >> brojevi[i];
    ull maksSegmenta = 0;
    for (int i = 0; i < n; i++) {
        ull xorSegmenta = brojevi[i];
        if (xorSegmenta > maksSegmenta)
            maksSegmenta = xorSegmenta;
        for (int j = i+1; j < n; j++) {
            xorSegmenta = xorSegmenta ^ brojevi[j];
            if (xorSegmenta > maksSegmenta)
                maksSegmenta = xorSegmenta;
        }
    }
    cout << maksSegmenta << endl;
    return 0;
}
```

Основни корак претходног алгоритма је да се за дату вредност текућег префикса пронађе онај претходни префикс који са датим даје највећу вредност XOR-а. Овај се проблем веома ефикасно може решити помоћу префиксног дрвета. Та техника је описана у задатку [Пар који даје највећи XOR](#).

На тај начин избегавамо унутрашњу петљу којом се анализира један по један префикс и укупну сложеност снижавамо на $O(n)$, при чему је константни фактор уз n једнак броју бита потребних за запис бројева.

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

using namespace std;

typedef unsigned long long ull;
```

```
struct Cvor {
    Cvor* grane[2];
};

Cvor* noviCvor() {
    Cvor* novi = new Cvor();
    novi->grane[0] = novi->grane[1] = nullptr;
    return novi;
}

void ubaci(Cvor* koren, ull broj) {
    Cvor* cvor = koren;
    ull mask = 1ull << (8*sizeof(ull) - 1);
    while (mask != 0) {
        int bit = (broj & mask) != 0;
        if (cvor->grane[bit] == nullptr)
            cvor->grane[bit] = noviCvor();
        cvor = cvor->grane[bit];
        mask >>= 1;
    }
}

ull maxXOR(Cvor* koren, ull broj) {
    Cvor* cvor = koren;
    ull rez = 0;
    ull mask = 1ull << (8*sizeof(ull) - 1);
    while (mask != 0) {
        int bit = (broj & mask) != 0;
        if (cvor->grane[!bit] != nullptr) {
            rez = rez | mask;
            cvor = cvor->grane[!bit];
        } else
            cvor = cvor->grane[bit];
        mask >>= 1;
    }
    return rez;
}

void obrisi(Cvor* koren) {
    if (koren != nullptr) {
        obrisi(koren->grane[0]);
        obrisi(koren->grane[1]);
        delete koren;
    }
}

int main() {
    int n;
    cin >> n;
    ull x;
    cin >> x;
    ull xorPrefiksa = x;
    Cvor* koren = noviCvor();
    ubaci(koren, xorPrefiksa);
    ull maxSegmenta = xorPrefiksa;
    for (int i = 1; i < n; i++) {
        ull x;
        cin >> x;
```

```

    xorPrefiksa = xorPrefiksa ^ x;
    maxSegmenta = max(maxSegmenta, xorPrefiksa);
    maxSegmenta = max(maxSegmenta, maxXOR(koren, xorPrefiksa));
    ubaci(koren, xorPrefiksa);
}
cout << maxSegmenta << endl;
obrisi(koren);
return 0;
}

```

5.2 Структура података за представљање дисјунктних подскупова (union-find)

Понекад је потребно одржавати у програму неколико дисјунктних подскупова одређеног скупа, при чему је моћи за дати елемент ефикасно пронаћи ком скупу припада (ова операција се зове `find`) и ефикасно спојити два задата подскупа у нови, већи подскуп (ту операцију називамо `union`). Помоћу операције `find` лако можемо за два елемента проверити да ли припадају истом подскупу тако што за сваки од њих пронађемо ознаку подскупа и проверимо да ли су оне једнаке.

Једна могућа имплементација је да се одржава пресликавање сваког елемента у ознаку подскупа којем припада. Ако претпоставимо да су сви елементи нумерисали бројевима од 0 до $n - 1$, онда ово пресликавање можемо реализовати помоћу обичног низа где се на позицији сваког елемента налази ознака подскупа којем он припада (ако елементи нису нумерисани бројевима, могли бисмо користити мапу уместо низа). Операција `find` је тада тривијална (само се из низа прочита ознака подскупа) и сложеност јој је $O(1)$. Операција `union` је много спорија, јер захтева да се ознаке свих елемената једног подскупа промене у ознаке другог, што захтева да се прође кроз читав низ и сложености је $O(n)$.

```

int id[MAX_N];
int n;

void inicijalizuj() {
    for (int i = 0; i < n; i++)
        id[i] = i;
}

int predstavnik(int x) {
    return id[x];
}

int u_istom_podskupu(int x, int y) {
    return predstavnik(x) == predstavnik(y);
}

void unija(int x, int y) {
    int idx = id[x], idy = id[y];
    for (int i = 0; i < n; i++)
        if (id[i] == idx)
            id[i] = idy;
}

```

Кључна идеја је да елементе не пресликавамо у ознаке подскупова, већ да подскупе чувамо у облику дрвета тако да сваки елемент сликамо у његовог родитеља у дрвету. Корене дрвета ћемо сликати саме у себе и сматрати их ознакама подскупова. Дакле, да бисмо на основу произвољног елемента сазнали ознаку подскупа ком припада, потребно је да прођемо кроз низ родитеља све док не стигнемо до корена. Нагласимо да су у овим дрветима показивачи усмерени од деце ка родитељима, за разлику од класичних дрвета где показивачи указују од родитеља ка деци.

Унију можемо вршити тако што корен једног подскупа усмеримо ка корену другог.

Први алгоритам одговара ситуацији у којој особа која промени адреду обавештава све друге особе о својој

5.2. СТРУКТУРА ПОДАКА ЗА ПРЕДСТАВЉАЊЕ ДИСЈУНКТНИХ ПОДСКУПОВА (UNION-FIND)

новој адреси. Други одговара ситуацији у којој само на старој адреси оставља информацију о својој новој адреси. Ово, наравно, мало успорава доставу поште, јер се мора прећи кроз низ преусмеравања, али ако тај низ није предугачак, може бити значајно ефикаснији од првог приступа.

```
int roditelj[MAX_N];
int n;

void inicijalizuj() {
    for (int i = 0; i < n; i++)
        roditelj[i] = i;
}

int predstavnik(int x) {
    while (roditelj[x] != x)
        x = roditelj[x];
    return x;
}

void unija(int x, int y) {
    int fx = predstavnik(x), fy = predstavnik(y);
    roditelj[fx] = fy;
}
```

Сложеност претходног приступа зависи од тога колико су дрвета којима се представљају подскупови уравнотежена. У најгорем случају се она могу издегенерирати у листу и тада је сложеност сваке од операција $O(n)$. Илуструјмо ово једним примером.

0 1 2 3 4 5 6 7

unija 7 6

0 1 2 3 4 5 6 6

unija 6 5

0 1 2 3 4 5 5 6

unija 5 4

0 1 2 3 4 4 5 6

unija 4 3

0 1 2 3 3 4 5 6

unija 3 2

0 1 2 2 3 4 5 6

unija 2 1

0 1 1 2 3 4 5 6

unija 1 0

0 0 1 2 3 4 5 6

Упит којим се тражи представник скупа којем припада елемент 7 се реализује низом корака којима се прелази преко следећих елемената 7,6,5,4,3,2,1,0. Иако се ово чини горим од претходног приступа, где је бар проналажење подскупа коштало $O(1)$ када су дрвета уравнотежена, тада је сложеност сваке од операција $O(\log n)$ и

централни задатак да би се на овој идеји изградила ефикасна структура података је да се на неки начин обезбеди да дрвета остану уравнотежена. Кључна идеја је да се приликом измена (а она се врше само у склопу операције уније), ако је могуће, обезбеди да се висина дрвета којим се представља унија не повећа у односу на висине појединачних дрвета која представљају скупе који се унирају (висину можемо дефинисати као број грана на путањи од тог чвора до њему најудаљенијег листа). Приликом прављења уније, имамо слободу избора корена ког ћемо усмерити ка другом корену. Ако се увек изабере да корен плићег дрвета усмеравамо ка дубљем, тада ће се висина уније повећавати само ако су оба дрвета кој унирамо исте висине. Висину дрвета можемо одржавати у посебном низу који ћемо из разлога који ће бити касније објашњени назвати `rang`.

```
int roditelj[MAX_N];
int n;
int rang[MAX_N];

void inicijalizuj() {
    for (int i = 0; i < n; i++) {
        roditelj[i] = i;
        rang[i] = 0;
    }
}

int predstavnik(int x) {
    while (roditelj[x] != x)
        x = roditelj[x];
    return x;
}

void unija(int x, int y) {
    int fx = predstavnik(x), fy = predstavnik(y);
    if (rang[fx] < rang[fy])
        roditelj[fx] = fy;
    else if (rang[fy] < rang[fx])
        roditelj[fy] = fx;
    else {
        roditelj[fx] = fy;
        rang[fy]++;
    }
}
```

Покажимо рад алгоритма на једном примеру. Подскупе ћемо представљати дрветима.

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

```
unija 1 2
```

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

```
unija 6 7
```

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

```
unija 4 7
```

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

```
unija 5 8
```

```
1 3      6      8
```

5.2. СТРУКТУРА ПОДАКА ЗА ПРЕДСТАВЉАЊЕ ДИСЈУНКТНИХ ПОДСКУПОВА (UNION-FIND)

2 4 7 5

uniја 1 3

1 6 8
2 3 4 7 5

uniја 5 4

1 6
2 3 4 7 8
 5

uniја 3 7

 6
1 4 7 8
2 3 5

Докажимо индукцијом да се у дрвету чији је корен на висини h налази бар 2^h чворова. База је почетни случај у коме је сваки чвор свој представник. Висина свих чворова је тада нула и сва дрвета имају $2^0 = 1$ чвор. Покажимо да свака унија одржава ову инваријанту. По индуктивној хипотези претпостављамо да оба дрвета која представљају подскупе који се унирају имају висине h_1 и h_2 и бар 2^{h_1} и 2^{h_2} чворова. Уколико се унирањем висина не повећа, инваријанта је очувана јер се број чворова повећао. Једини случај када се повећава висина уније је када је $h_1 = h_2$ и тада обједињено дрво има висину $h = h_1 + 1 = h_2 + 1$ и бар $2^{h_1} + 2^{h_2} = 2^h$ чворова. Тиме је тврђење доказано. Дакле, сложеност сваке операције проналаска представника у скупу од n чворова је $O(\log n)$, а пошто унирање након проналажења представника врши још само $O(1)$ операција, и сложеност налажења уније је $O(\log n)$.

Рецимо и да је уместо висине могуће одржавати број чворова у сваком од подскупова. Ако увек усмеравамо представника мањег ка представнику већег подскупа, поново ћемо добити логаритамску сложеност најгорег случаја за обе операције. Ово важи зато што и овај начин прављења уније гарантује да не можемо имати високо дрво са малим бројем чворова. Да би се добило дрво висине 1, потребна су најмање два чвора; да би се добило дрво висине 2 најмање четири чвора (јер се спајају два дрвета висине 1 која имају бар по два чвора). Да би се добило дрво висине h потребно је најмање 2^h чворова. Одавде следи да ће висине свих дрвета у овој структури бити висине $O(\log n)$.

Иако је ова сложеност сасвим прихватљива (сложеност проналажења n унија је $O(n \log n)$), може се додатно побољшати врло једноставном техником познатом као *компресија путања*. Наиме, приликом проналажења представника можемо све чворове кроз које пролазимо усмерити према корену. Један начин да се то постигне је да се након проналажења корена, поново прође кроз низ родитеља и сви показивачи усмере према корену.

```
int predstavnik(int x) {
    int koren = x;
    while (koren != roditelj[koren])
        koren = roditelj[koren];
    while (x != koren) {
        int tmp = roditelj[x];
        roditelj[x] = koren;
        x = tmp;
    }
    return koren;
}
```

За све чворове који се обилазе од полазног чвора до корена, дужине путања до корена се након овога смањују на 1, међутим, као што примећујемо, низ рангова се не мења. Ако рангове тумачимо као број чворова у подскупу, онда се приликом компресије путање та статистика и не мења, па је поступак коректан. Ако рангове тумачимо као висине, јасно је да приликом компресије путање низ висина постаје неажуран. Међутим, интересантно је да ни у овом случају нема потребе да се он ажурира. Наиме, бројеви који се сада чувају у том низу не представљају више висине чворова, већ горње границе висина чворова. Ови бројеви се надаље сматрају ранговима чворова, тј. помоћним подацима који нам помажу да преусмеримо чворове приликом

унирања. Показује се да се овим не нарушава сложеност најгорег случаја и да функција наставља коректно да ради.

У претходној имплементацији се два пута пролази кроз путању од чвора до корена. Ипак, сличне перформансе се могу добити и само у једном пролазу. Постоје два начина на који се ово може урадити: један од њих је да се сваки чвор усмери ка родитељу свог родитеља. За све чворове који се обилазе од полазног чвора до корена, дужине путања до корена се након овога смањују двоструко, што је довољно за одличне перформансе.

```
int predstavnik(int x) {
    while (x != roditelj[x]) {
        tmp = roditelj[x];
        roditelj[x] = roditelj[roditelj[x]];
        x = tmp;
    }
    return x;
}
```

Други начин подразумева да се приликом проласка од чвора ка корену сваки други чвор на путањи усмери ка родитељу свог родитеља.

```
int predstavnik(int x) {
    while (x != roditelj[x]) {
        roditelj[x] = roditelj[roditelj[x]];
        x = roditelj[x];
    }
    return x;
}
```

Приметимо да је овим додата само једна линија кода у првобитну имплементацију. Овом једноставном променом амортизована сложеност операција постаје само $O(\alpha(n))$, где је $\alpha(n)$ инверзна Акерманова функција која страшно споро расте. За били који број n који је мањи од броја атома у целом универзуму важи да је $\alpha(n) < 5$, тако да је време практично константно.

Задатак: Први пут кроз матрицу

Логичка матрица димензије $n \times n$ у почетку садржи све нуле. Након тога се насумично додаје једна по једна јединица. Кретање по матрици је могуће само по јединицама и то само на доле, на горе, на десно и на лево. Написати програм који учитава димензију матрице, а затим позицију једне по једне јединице и одређује након колико њих је први пут могуће сићи од врха до дна матрице (са произвољног поља прве врсте до произвољног поља последње врсте матрице).

Улаз: Са стандардног улаза се учитава димензија матрице $1 \leq n \leq 200$, затим број поља m ($1 \leq m \leq n^2$) у које се уписује јединица, а затим у наредних m редова координате тих поља (број врсте и број колоне од 0 до $n - 1$, раздвојени размаком).

Издаз: Опис излазних података.

Пример

Улаз	Издаз
4	8
9	
0 0	
0 1	
1 1	
3 3	
1 3	
2 0	
3 0	
2 1	
2 2	

Објашњење

5.2. СТРУКТУРА ПОДАКА ЗА ПРЕДСТАВЉАЊЕ ДИСЈУНКТНИХ ПОДСКУПОВА (UNION-FIND)

После 8 учитаних поља, матрица постаје

```
1100
0101
1100
1001
```

и врх и дно постају спојени.

Решење

Основна идеја је да се формирају сви подскупови елемената између којих постоји пут (они формирају тзв. компоненте повезаности). Сваки пут када се успостави веза између нека два елемента таква два подскупа, подскупови се спајају. Провера да ли постоји пут између два елемента своди се онда на проверу да ли они припадају истом подскупу.

Путања од врха до дна постоји ако постоји путања од било ког елемента у првој врсти матрице до било ког елемента у дну матрице. То би довело до тога да у сваком кораку морамо да проверавамо све парове елемената из горње и доње врсте. Међутим, можемо и боље. Додаћемо вештачки почетни чвор (назовимо га извор) и спојићемо га са свим чворовима у првој врсти матрице и завршни чвор (назовимо га ушће) и спојићемо га са свим чворовима у последњој врсти матрице. Тада се у сваком кораку само може проверити да ли су извор и ушће спојени тј. да ли припадају истом подскупу.

Подскупове можемо чувати помоћу структуре података за представљање дисјунктних подскупова (енгл. union-find).

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

using namespace std;

vector<int> UF_roditelj;
vector<int> UF_rang;

void UF_inicijalizacija(int n) {
    UF_roditelj.resize(n);
    UF_rang.resize(n);
    for (int i = 0; i < n; i++) {
        UF_roditelj[i] = i;
        UF_rang[i] = 0;
    }
}

int UF_predstavnik(int x) {
    while (x != UF_roditelj[x]) {
        UF_roditelj[x] = UF_roditelj[UF_roditelj[x]];
        x = UF_roditelj[x];
    }
    return x;
}

void UF_unija(int x, int y) {
    int fx = UF_predstavnik(x), fy = UF_predstavnik(y);
    if (UF_rang[fx] < UF_rang[fy])
        UF_roditelj[fx] = fy;
    else if (UF_rang[fy] < UF_rang[fx])
        UF_roditelj[fy] = fx;
    else {
        UF_roditelj[fx] = fy;
        UF_rang[fy]++;
    }
}
```

```

// redni broj elementa (x, y) u matrici
int kod(int x, int y, int n) {
    return x*n + y;
}

int main() {
    // dimenzija matrice
    int n;
    cin >> n;

    // alociramo matricu n*n
    vector<vector<bool>> a(n);
    for (int i = 0; i < n; i++)
        a[i].resize(n, false);

    // dva dodatna veštačka čvora
    const int izvor = n*n;
    const int usce = n*n+1;

    // inicijalizujemo union-find strukturu za sve elemente matrice
    // (njih n*n), izvor i ušće
    UF_inicijalizacija(n*n + 2);

    // spajamo izvor sa svim elementima u prvoj vrsti matrice
    for (int i = 0; i < n; i++)
        UF_unija(izvor, kod(0, i, n));

    // spajamo sve elemente u poslednjoj vrsti matrice sa ušćem
    for (int i = 0; i < n; i++)
        UF_unija(kod(n-1, i, n), usce);

    // broj jedinica
    int m;
    cin >> m;

    // korak u kom se spajaju izvor i usce
    int korak = -1;

    // učitavamo i obrađujemo jednu po jednu jedinicu
    for (int k = 1; k <= m; k++) {
        int x, y;
        cin >> x >> y;
        // ako je u matrici već jedinica, nema šta da se radi
        if (a[x][y]) continue;
        // upisujemo jedinicu u matricu
        a[x][y] = true;
        // povezujemo podskupove u sva četiri smera
        if (x > 0 && a[x-1][y])
            UF_unija(kod(x, y, n), kod(x-1, y, n));
        if (x + 1 < n && a[x+1][y])
            UF_unija(kod(x, y, n), kod(x+1, y, n));
        if (y > 0 && a[x][y-1])
            UF_unija(kod(x, y, n), kod(x, y-1, n));
        if (y + 1 < n && a[x][y+1])
            UF_unija(kod(x, y, n), kod(x, y+1, n));
        // proveravamo da li su izvor i ušće spojeni
        if (UF_predstavnik(izvor) == UF_predstavnik(usce)) {

```

5.2. СТРУКТУРА ПОДАКА ЗА ПРЕДСТАВЉАЊЕ ДИСЈУНКТНИХ ПОДСКУПОВА (UNION-FIND)

```
        korak = k;
        break;
    }
}

cout << korak << endl;

return 0;
}
```

Задатак: Распоред са максималним збиром профита

Дати су послови који сви трају јединично време и за сваки посао је познато који је крајњи рок да се заврши и колики је профит ако се тај посао заврши у року. Напиши програм који одређује максимални профит који се може остварити.

Улаз: Са стандардног улаза се учитава број послова n ($1 \leq n \leq 50000$), а затим за сваки посао рок завршетка (природан број од 1 до n) и профит (цео број од 1 до 100).

Издаз: На стандардни издаз исписати максимални профит.

Пример

Улаз	Издаз
6	290
3 40	
3 80	
4 30	
4 100	
1 70	
1 60	

Решење

Задатак решавамо грамзивим алгоритмом. Послове распоређујемо у опадајућем редоследу профита. Посао са највећим профитом распоређујемо у најкаснији термин у ком је то могуће (пре истека његовог рока завршетка).

Докажимо коректност претходне грамзиве стратегије. Претпоставимо да се посао са највећим профитом не налази у оптималном распореду. Ако постоји посао у распореду који се завршава пре рока завршетка посла са највећим профитом тада уместо њега можемо ставити посао са највећим профитом и тако добити већи профит, што је контрадикција са претпоставком оптималности. Дакле, посао са највећим профитом мора бити део распореда (ако постоји било који празан термин у коме се он може распоредити). У оптималном распореду он може бити распоређен у последњем термину у ком је то могуће. Наиме, ако је он распоређен пре тога, а тај термин је празан, просто га можемо извршити касније. Ако се у том термину налази неки други посао, онда та два посла можемо просто распоредити, без промене укупног профита.

Имплементација овог поступка тече тако што послове сортирамо опадајуће по профиту, а затим за сваки посао са листе тражимо најкаснији термин у коме је могуће распоредити га. Ако такав термин не постоји, посао прескачемо, а у супротном га распоређујемо баш у том термину. Критично место је претрага термина за текући посао.

Оно што прво пада на памет је одржавање низа логичких вредности којима бележимо који је термин слободан, а који није. Слободан термин одређујемо линеарном претрагом тако што крећемо од термина непосредно пре рока завршетка и померамо се ка раније, све док су термин заузети. Та претрага може бити линеарне сложености (ако су сви термин заузети), што може довести до алгоритма чија је сложеност $O(m \cdot n)$ где је m број термина (најкаснији рок завршетка посла) док је n укупан број послова.

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

using namespace std;
```

```

int main() {
    int n;
    cin >> n;
    vector<pair<int, int>> poslovi(n);
    for (int i = 0; i < n; i++)
        cin >> poslovi[i].first >> poslovi[i].second;

    sort(begin(poslovi), end(poslovi),
        [](const auto& p1, const auto& p2) {
            return p1.second > p2.second;
        });

    int profit = 0;
    int brojRasporedjenih = 0;
    int maksRok = max_element(begin(poslovi), end(poslovi))->first;
    vector<bool> zauzeto(maksRok, false);
    for (const auto& posao : poslovi) {
        int vreme = posao.first - 1;
        while (vreme >= 0 && zauzeto[vreme])
            vreme--;
        if (vreme >= 0) {
            zauzeto[vreme] = true;
            profit += posao.second;
            brojRasporedjenih++;
            if (brojRasporedjenih == maksRok)
                break;
        }
    }

    cout << profit << endl;

    return 0;
}

```

Боље решење се постиже ако се употреби идеја структуре података за ефикасно одређивање унија. Наиме, можемо креирати скупове термина такве да се у сваком скупу налазе групе повезаних термина тј. тако да сви термини у истом скупу имају заједничког најкаснијег слободног претходника. Крећемо од тога да су сви скупови једночлани тј. да је за сваки жељени термин он сам термин у коме се посао може завршити (сваки термин бива постављен као представник свог скупа). Када се неки посао закаже, тада се скуп коме је он припадао спаја са скупом који се налази непосредно испред термина у коме је заказан. Да бисмо могли да региструјемо и термине у којима више није могуће заказивати часове уводимо и посебан термин 0 (и третирамо га као и све остале).

Прикажимо на примеру како можемо распоредити следеће послове

```

a: 3 40
b: 3 80
c: 4 30
d: 4 100
e: 1 70
f: 1 60

```

Након сортирања по профитима добијамо следећи редослед.

```

d: 4 100
b: 3 80
e: 1 70
f: 1 60
a: 3 40
c: 4 30

```

5.2. СТРУКТУРА ПОДАКА ЗА ПРЕДСТАВЉАЊЕ ДИСЈУНКТНИХ ПОДСКУПОВА (UNION-FIND)

Крећемо од следећег стања низа родитеља и распоређених послова.

```
0 1 2 3 4
0 1 2 3 4
- - - - -
```

Посао d се може распоредити у термин 4 чиме добијамо:

```
0 1 2 3 4
0 1 2 3 3
- - - - d
```

Посао b се може распоредити у термин 3 чиме добијамо:

```
0 1 2 3 4
0 1 2 2 3
- - - b d
```

Посао e се може распоредити у термин 1 чиме добијамо:

```
0 1 2 3 4
0 0 2 2 3
- e - b d
```

Посао f се не може распоредити (пошто је представник његовог термина 1 сада једнак 0), па га прескачемо. Посао a распоређујемо у термин 2 (што је тренутни представник његовог термина 3) чиме добијамо

```
0 1 2 3 4
0 0 0 2 3
- e f b d
```

Посао e не можемо распоредити (јер је представник његовог термина нула). Ако радимо компресију путање, долазимо у стање

```
0 1 2 3 4
0 0 0 0 3
- e f b d
```

На крају ни посао f не можемо распоредити (јер је представник његовог термина нула). Ако радимо компресију путање, долазимо у стање

```
0 1 2 3 4
0 0 0 0 0
- e f b d
```

Приметимо да смо поступак распоређивања могли прекинути када је број распоређених послова достигао број укупан број термина.

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

```
using namespace std;
```

```
vector<int> UF_roditelj;
```

```
void UF_inicijalizacija(int n) {
    UF_roditelj.resize(n);
    for (int i = 0; i < n; i++)
        UF_roditelj[i] = i;
}
```

```
int UF_predstavnik(int a) {
    while (UF_roditelj[a] != a) {
        UF_roditelj[a] = UF_roditelj[UF_roditelj[a]];
    }
}
```

```

    a = UF_roditelj[a];
}
return a;
}

void UF_unija(int a, int b) {
    UF_roditelj[a] = b;
}

int main() {
    int n;
    cin >> n;
    vector<pair<int, int>> poslovi(n);
    for (int i = 0; i < n; i++)
        cin >> poslovi[i].first >> poslovi[i].second;

    sort(begin(poslovi), end(poslovi),
        [](const auto& p1, const auto& p2) {
            return p1.second > p2.second;
        });

    int profit = 0;
    int brojRasporedjenih = 0;
    int maksRok = max_element(begin(poslovi), end(poslovi))->first;

    UF_inicijalizacija(maksRok + 1);
    for (const auto& posao : poslovi) {
        int vreme = UF_predstavnik(posao.first);
        if (vreme > 0) {
            profit += posao.second;
            UF_unija(vreme, UF_predstavnik(vreme - 1));
            brojRasporedjenih++;
            if (brojRasporedjenih == maksRok)
                break;
        }
    }

    cout << profit << endl;

    return 0;
}

```

5.3 Упити распона

Одређене структуре података су посебно погодне за проблеме у којима се тражи да се над низом елемената извршавају упити који захтевају израчунавање статистика неких сегмената тј. распона низа (енгл. range queries). Најчешће се посматрају зборови елемената сегмената, али могуће је разматрати и минимум, максимум, производ и неке друге операције. У зависности од тога да ли се низ мења између извршавања упита или се упити извршавају над низом који је стално исти разликујемо *статичке упите распона* и *динамичке упите распона*. Статички упити распона се често могу решити прилично елементарним техникама (одржавањем низа зборова префикса или низа разлика суседних елемената низа), док динамички упити распона захтевају коришћење напреднијих структура података (сегментних стабала, Фенвикових стабала).

5.3.1 Статички упити распона

Илуструјмо статичке упите распона кроз неколико једноставних задатака.

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	15
1 2 3 4 5	9
3	3
0 4	
1 3	
2 2	

Решење**Директно решење**

Директно решење подразумева да све бројеве учитамо у низ, а затим да за сваки упит изнова рачунамо збир одговарајућег сегмента низа.

Сложеност оваквог приступа је $O(nm)$.

Пошто се фаза учитавања и исписа података преплићу, потребно је прилагодити програм аутоматском тестирању (коришћењем `cin.tie(0)` и `\n` уместо `endl`), но то не може поправити неефикасност овог наивног алгорита.

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

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

    int n;
    cin >> n;
    vector<int> brojevi(n);
    for (int i = 0; i < n; i++)
        cin >> brojevi[i];
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        int zbir = 0;
        for (int j = a; j <= b; j++)
            zbir += brojevi[j];
        cout << zbir << '\n';
    }
    return 0;
}
```

Збирови префикса

Једноставно ефикасно решење је засновано на наредној идеји: уместо чувања елемената низа, можемо чувати низ збирова префикса низа. Збир сваког сегмента $[l, d]$ можемо разложити на разлику збира префикса до

елемента d и префикса до елемента $l - 1$. Ако користимо ознаку $\sum_{k=m}^n a_k$ која означава збир $a_m + a_{m+1} + \dots + a_n$, можемо записати да је

$$\sum_{k=l}^d a_k = \sum_{k=0}^d a_k - \sum_{k=0}^{l-1} a_k.$$

Зборови свих префикса се могу израчунати и сместити у додатни (а ако је уштеда меморије битна, онда чак и у оригинални) низ. Дакле, током учитавања елемената можемо формирати низ збиров префикса (рачунаћемо их инкрементално, јер се сваки наредни збир префикса добија увећавањем претходног збира префикса за текући елемент низа). Нека z_i означава збир елемената префикса одређеног позицијама из интервала $[0, i]$. Формирамо, дакле, низ $z_i = \sum_{k=0}^{i-1} a_k$ (при чему је $z_0 = 0$, збир празног префикса). Тада збир елемената у сегменту позиција $[l, d]$ израчунавамо као $z_{d+1} - z_l$.

За учитавање бројева и формирање низа збиров префикса потребно нам је $O(n)$ корака. Након оваквог претпроцесирања, збир сваког сегмента се може израчунати у времену $O(1)$, па је укупна сложеност $O(n+m)$.

Пошто се у овом задатку преплићу фаза учитавања и фаза исписа података на стандардни улаз и излаз, потребно је обратити пажњу на неефикасност која настаје због честог пражњења излазног бафера. Потребно је развезати `cin` и `cout` коришћењем `cin.tie(0)` и уместо помоћу `endl` у нови ред прелазити помоћу `\n`. Наравно, ово има смисла само у случају аутоматске примене програма на велике улазе и излазе - овим изменама програм престаје да ради коректно у интерактивном режиму.

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

using namespace std;

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

    int n;
    cin >> n;
    vector<int> zbrovi_prefiksa(n+1);

    zbrovi_prefiksa[0] = 0;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        zbrovi_prefiksa[i+1] = zbrovi_prefiksa[i] + x;
    }

    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        cout << zbrovi_prefiksa[b+1] - zbrovi_prefiksa[a] << '\n';
    }

    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.

5.3. УПИТИ РАСПОНА

Улаз: Са стандардног улаза се уноси број километара N ($10 \leq N \leq 10000$), затим, у наредном реду, број предмета M ($0 \leq M \leq 10000$), а након тога, у наредних M редова по три цела броја раздвојена размацама 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).

Изаз: На стандардни излаз исписати масу терета у килограмима на сваком километру пута (иза сваке масе написати по један размак).

Пример

Улаз	Изаз
10	0 10 25 35 35 25 25 15 0
3	
1 5 10	
3 7 10	
2 8 15	

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

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 $R_0 + R_1 + \dots + R_i = M_0 + (M_1 - M_0) + \dots + (M_i - M_{i-1}) = M_i$.

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-Лајбницевој 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: 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 $1 \leq l \leq d \leq n$).

Изаз: На стандардни излаз исписати највећу могућу укупну количину светлости коју могу добити сви станови.

Пример 1

Улаз Излаз
3 25
5 3 2
3
1 2
2 3
1 3

Пример 2

Улаз Излаз
5 33
5 2 4 1 3
3
1 5
2 3
2 3

Решење

Укупно осветљење се може израчунати као

$$\sum_{i=1}^s \sum_{j=l_i}^{d_i} a_j$$

Исти резултат се може добити и на другачији начин, као

$$\sum_{j=1}^n m_j \cdot a_j,$$

где је m_j број сегмената $[l_i, d_i]$ који садрже вредност j .

Да би збир био максималан, потребно је да уз веће вредности m_j стоје већи бројеви a_j . Заиста, претпоставимо да у максималном збиру учествују парови $m_x a_x + m_y a_y$, да је $m_x > m_y$, али да је $a_x < a_y$. Разменом вредности a_x и a_y добија се већи збир, што је контрадикција. Заиста $(m_x a_y + m_y a_x) - (m_x a_x + m_y a_y) = m_x (a_y - a_x) - m_y (a_y - a_x) = (m_x - m_y)(a_y - a_x) > 0$.

Зато максимални збир можемо лако израчунати сортирањем вредности низова $m_1, \dots, m_n$ и $a_1, \dots, a_n$ и израчунавањем њиховог скаларног производа.

Остаје још питање како ефикасно израчунати вредности низа $m_1, \dots, m_n$. Директно бројање сегмената би било веома неефикасно. Најједноставније решење је да употребимо низ разлика низа m (слично као у задатку **Увећавање сегмената**).

Низ $m_1, \dots, m_n$ можемо конструисати у времену $O(s + n)$. Након тога временом доминира сортирање (два пута по $O(n \log n)$), док је за израчунавање скаларног производа потребно $O(n)$ операција.

5.3.2 Сегментна дрвета

Видели смо да низ збирова префикса, омогућава ефикасно постављање упита над сегментима низа, али не омогућава ефикасно ажурирање елемената низа, јер је потребно ажурирати све збирове префикса након ажурираног елемента, што је нарочито неефикасно када се ажурирају елементи близу почетка низа (сложеност најгорег случаја је $O(n)$). Низ разлика суседних елемената допушта стална ажурирања низа, међутим, извршавање упита читавања стања низа подразумева реконструкцију низа, што је сложености $O(n)$.

Разматраћемо проблеме у којима је потребно да се упити ажурирања низа и читавања његових статистика јављају испреплетано. За разлику од претходних, статичких упита над распонима (енгл. *static range queries*), овде ћемо разматрати тзв. *динамичке упити над распонима* (енгл. *dynamic range queries*), тако да је потребно развити напредније структуре података које омогућавају извршавање оба типа упита ефикасно. На пример, размотримо проблем имплементације структуре података која обезбеђује ефикасно израчунавање збирова сегмената датог низа одређених интервалима позиција $[a, b]$, при чему се појединачни елементи низа могу често мењати.

У наставку ћемо видети две различите, али донекле сличне структуре података које дају ефикасно решење претходног проблема и њему сличних.

Једна структура података која омогућава прилично једноставно и ефикасно решавање овог проблема су *сегментна дрвета*. Опет се током фазе препроцесирања израчунавају зборови одређених сегмената полазног низа, а онда се збир елемената произвољног сегмента полазног низа изражава у функцији тих унапред израчунатих збирова. Рецимо и да сегментна дрвета нису специфична само за сабирање, већ се могу користити и за друге статистике сегмената које се израчунавају асоцијативним операцијама (на пример за одређивање најмањег или највећег елемента, нзд-а свих елемената и слично).

Претпоставимо да је дужина низа степен броја 2 (ако није, низ се може допунити до најближег степена броја 2, најчешће нулама). Чланови низа представљају листове дрвета. Групишемо два по два суседна чвора и на сваком наредном нивоу дрвета чувамо родитељске чворове који чувају збирове своја два детета. Ако је дат низ 3, 4, 1, 2, 6, 5, 1, 4, сегментно дрво за збирове изгледа овако.

```

          26
        10  16
       7   3  11  5
      3  4  1  2  6  5  1  4

```

Пошто је дрво потпуно, најједноставнија имплементација је да се чува имплицитно у низу (слично као у случају хипа). Претпоставићемо да елементе дрвета смештамо од позиције 1, јер је тада аритметика са индексима мало једноставнија (елементи полазног низа могу бити индексирани класично, кренувши од нуле).

```

0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
- 26 10 16 7  3  11 5  3  4  1  2  6  5  1  4

```

Уочимо неколико карактеристика овог начина смештања. Корен је смештен на позицији 1. Елементи полазног низа налазе се на позицијама $[n, 2n - 1]$. Елемент који се у полазном низу налази на позицији p , се у сегментном дрвету налази на позицији $p + n$. Лево дете чвора k налази се на позицији $2k$, а десно на позицији $2k + 1$. Дакле, на парним позицијама се налазе лева деца својих родитеља, а на непарним десна. Родитељ чвора k налази се на позицији $\lfloor \frac{k}{2} \rfloor$.

Размотримо сада како бисмо нашли збир елемената на позицијама из сегмента $[2, 6]$, тј. збир елемената 1, 2, 6, 5, 1. У сегментном дрвету тај сегмент је смештен на позицијама $[2 + 8, 6 + 8] = [10, 14]$. Збир прва два елемента (1, 2) се налази у чвору изнад њих, збир наредна два елемента (6, 5) такође, док се у родитељском чвору елемента 1 налази његов збир са елементом 4, који не припада сегменту који сабирамо. Зато збир елемената на позицијама $[10, 14]$ у сегментном дрвету можемо разложити на збир елемената на позицијама $[5, 6]$ и елемента на позицији 14.

Размотримо и како бисмо рачунали збир елемената на позицијама из сегмента $[3, 7]$, тј. збир елемената 2, 6, 5, 1, 4. У сегментном дрвету тај сегмент је смештен на позицијама $[3 + 8, 7 + 8] = [11, 15]$. У родитељском чвору елемента 2 налази се његов збир са елементом 1 који не припада сегменту који сабирамо. Зборови елемената 6 и 5 и елемената 1 и 4 се налазе у чворовима изнад њих, а збир сва четири елемента у чвору изнад њих.

Уместо операције којом се мења члану низа на позицији i додељује вредност v , често се разматра функција која елемент низа на позицији i полазног низа увећава за дату вредност v и у складу са тим ажурира сегментно дрво. Свака од ове две функције се лако изражава преко оне друге.

Имплементација сегментног дрвета за друге асоцијативне операције је скоро идентична, осим што се оператор $+$ мења другом операцијом.

Задатак: Суме сегмената променљивог низа

Напиши програм који израчунава збирове датих сегмената низа (поднизова узастопних елемената), при чему се током рада програма поједини елементи низа могу мењати.

Улаз: У првој линији стандардног улаза налази се број n ($1 \leq n \leq 100000$), а у наредној линији низ од n елемената (елементи су цели бројеви између 0 и 10, раздвојени са по једним размаком). У наредној линији налази се број m ($1 \leq m \leq 100000$), а у наредних m линија упити. Подржане су две врсте упита:

- $p\ i\ v$ – извршавање овог упита подразумева да се у низ на позицију i упише вредност v ($0 \leq i < n$, $0 \leq v \leq 10$).
- $z\ a\ b$ – извршавање овог упита подразумева да се израчуна и на стандардни излаз испише збир елемената низа који су на позицијама $[a, b]$.

Излаз: Стандардни излаз садржи резултате z упита (сваки у посебној линији).

Пример

Улаз	Излаз
5	15
1 2 3 4 5	7
5	19
z 0 4	
z 2 3	
p 2 5	
p 3 6	
z 0 4	

Решење

Решење грубом силом подразумева да се одржава низ, да се упити p извршавају директном променом елемената низа (то се ради у $O(1)$), а да се упити z извршавају сабирањем елемената низа (то се ради у времену $O(n)$). Укупна сложеност таквог приступа је $O(mn)$.

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

```
using namespace std;
```

```
int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);
    int n;
    cin >> n;
    vector<int> niz(n);
    for (int i = 0; i < n; i++)
        cin >> niz[i];
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        string upit;
        cin >> upit;
        if (upit == "p") {
            int i, v;
            cin >> i >> v >> ws;
            niz[i] = v;
        } else if (upit == "z") {
            int a, b;
            cin >> a >> b;
            cout << accumulate(next(begin(niz), a), next(begin(niz), b+1), 0) << '\n';
        }
    }
    return 0;
}
```

Задатак можемо ефикасније решити ако збирове одржавамо у сегментном дрвету. Операције над дрветом можемо реализовати навише (од листова према корену).

Формирање сегментног дрвета на основу датог низа је веома једноставно. Прво се елементи полазног низа прекопирају у дрво, кренувши од позиције n . Затим се сви унутрашњи чворови дрвета (од позиције $n - 1$, па уназад до позиције 1) попуњавају као зборови своје деце (на позицију k уписујемо збир елемената на позицијама $2k$ и $2k + 1$). Сложеност ове операције је очигледно линеарна у односу на дужину низа n .

За све унутрашње елементе сегмента смо сигурни да се њихов збир налази у чворовима изнад њих. Једини изузетак могу да буду елементи на крајевима сегмента. Ако је елемент на левом крају сегмента лево дете (што је еквивалентно томе да се налази на парној позицији) тада се у његовом родитељском чвору налази његов збир са елементом десно од њега који такође припада сегменту који треба сабрати (осим евентуално у случају једночланог сегмента). У супротном (ако се налази на непарној позицији), у његовом родитељском чвору је његов збир са елементом лево од њега, који не припада сегменту који сабирамо. У тој ситуацији, тај елемент ћемо посебно додати на збир и искључити из сегмента који сабирамо помоћу родитељских чворова. Ако је елемент на десном крају сегмента лево дете (ако се налази на парној позицији), тада се у његовом родитељском чвору налази његов збир са елементом десно од њега, који не припада сегменту који сабирамо. И у тој ситуацији, тај елемент ћемо посебно додати на збир и искључити из сегмента који сабирамо помоћу родитељских чворова. На крају, ако се крајњи десни елемент налази у десном чвору (ако је на непарној позицији), тада се у његовом родитељском чвору налази његов збир са елементом лево од њега који припада сегменту који сабирамо (осим евентуално у случају једночланог сегмента). Пошто се у сваком кораку дужина сегмента $[a, b]$ полови, а она је у почетку сигурно мања или једнака n , сложеност ове операције је $O(\log n)$.

Приликом ажурирања неког елемента потребно је ажурирати све чворове на путањи од тог листа до корена. С обзиром да знамо позицију родитеља сваког чвора и ова операција се може веома једноставно имплементирати. Пошто се k полови у сваком кораку петље, а крећа од вредности највише $2n - 1$, и сложеност ове операције је $O(\log n)$.

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

using namespace std;

// najmanji stepen dvojke veci ili jednak od n
int stepenDvojke(int n) {
    int s = 1;
    while (s < n)
        s <= 1;
    return s;
}

// na osnovu datog niza a dužine n
// u kom su elementi smešteni od pozicije 0
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije 1
vector<int> formirajDrvo(const vector<int>& a) {
    int n = stepenDvojke(a.size());
    vector<int> drvo(2*n, 0);
    // kopiramo originalni niz u listove
    copy(begin(a), end(a), next(begin(drvo), n));
    // ažuriramo roditelje već upisanih elemenata
    for (int k = n-1; k >= 1; k--)
        drvo[k] = drvo[2*k] + drvo[2*k+1];
    return drvo;
}

// izračunava se zbir elemenata polaznog niza dužine n koji se
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drвета
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
```

```

int zbirSegmenta(const vector<int>& drvo, int a, int b) {
    int n = drvo.size() / 2;
    a += n; b += n;
    int zbir = 0;
    while (a <= b) {
        if (a % 2 == 1) zbir += drvo[a++];
        if (b % 2 == 0) zbir += drvo[b--];
        a /= 2;
        b /= 2;
    }
    return zbir;
}

// ažurira segmentno drvo smešteno u niz drvo od pozicije 1
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void postavi(vector<int>& drvo, int i, int v) {
    int n = drvo.size() / 2;
    // prvo ažuriramo odgovarajući list
    int k = i + n;
    drvo[k] = v;
    // ažuriramo sve roditelje izmenjenih čvorova
    for (k /= 2; k >= 1; k /= 2)
        drvo[k] = drvo[2*k] + drvo[2*k+1];
}

int main() {
    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];
    vector<int> drvo = formirajDrvo(a);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        string upit;
        cin >> upit;
        if (upit == "p") {
            int i, v;
            cin >> i >> v >> ws;
            postavi(drvo, i, v);
        } else if (upit == "z") {
            int a, b;
            cin >> a >> b;
            cout << zbirSegmenta(drvo, a, b) << '\n';
        }
    }
    return 0;
}

```

Претходни приступ формира дрво одоздо навише (прво се попуне листови, па онда корен). Још један начин је да се дрво формира рекурзивно, одозго наниже. Иако је ова имплементација компликованија и мало неефикаснија, приступ одозго наниже је у неким каснијим операцијама неизбежан, па га илуструјемо на овом једноставном примеру. Сваки чвор дрвета представља збир одређеног сегмента позиција полазног низа. Сегмент је једнозначно одређен позицијом k у низу који одговара сегментном дрвету, али да бисмо олакшали имплементацију границе тог сегмента можемо кроз рекурзију прослеђивати као параметар функције, заједно

са вредношћу k (нека је то сегмент $[x, y]$). Дрво крећемо да градимо од корена где је $k = 1$ и $[x, y] = [0, n - 1]$. Ако родитељски чвор покрива сегмент $[x, y]$, тада лево дете покрива сегмент $[x, \lfloor \frac{x+y}{2} \rfloor]$, а десно дете покрива сегмент $[\lfloor \frac{x+y}{2} \rfloor + 1, y]$. Дрво попуњавамо рекурзивно, тако што прво попуњимо лево поддрво, затим десно и на крају вредност у корену израчунавамо као Дрво попуњавамо рекурзивно, тако што прво попуњимо лево поддрво, затим десно и на крају вредност у корену израчунавамо као збир вредности у левом и десном детету. Излаз из рекурзије представљају листови, које препознајемо по томе што покривају сегменте дужине 1, и у њих само копирамо елементе са одговарајућих позиција полазног низа.

И за операцију сабирања можемо направити и рекурзивну имплементацију која врши израчунавање одозго наниже. За сваки чвор у сегментном дрвету функција враћа колики је допринос сегмента који одговара том чвору и његовим наследницима траженом збиру елемената на позицијама из сегмента $[a, b]$ у полазном низу. На почетку крећемо од корена и рачунамо допринос целог дрвета збиру елемената из сегмента $[a, b]$. Постоје три различита могућа односа између сегмента $[x, y]$ који одговара текућем чвору и сегмента $[a, b]$ чији збир елемената тражимо. Ако су дисјунктни, допринос текућег чвора збиру сегмента $[a, b]$ је нула. Ако је $[x, y]$ у потпуности садржан у $[a, b]$, тада је допринос потпун, тј. цео збир сегмента $[x, y]$ (а то је број уписан у низу на позицији k) доприноси збиру елемената на позицијама из сегмента $[a, b]$. На крају, ако се сегменти секу, тада је допринос текућег чвора једнак збиру доприноса његовог левог и десног детета. Иако није сасвим очигледно, и ова процедура ће имати сложеност $O(\log n)$.

И операцију промене вредности елемената можемо имплементирати одозго наниже. Сложеност је опет $O(\log n)$, јер се дужина интервала $[x, y]$ у сваком кораку бар два пута смањује.

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

using namespace std;

// od elemenata niza a sa pozicija [x, y]
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije k
void formirajDrvo(const vector<int>& a, vector<int>& drvo,
                  size_t k, size_t x, size_t y) {
    if (x == y)
        // u listove prepisujemo elemente polaznog niza
        drvo[k] = x < a.size() ? a[x] : 0;
    else {
        // rekurzivno formiram levo i desno poddrvo
        int s = (x + y) / 2;
        formirajDrvo(a, drvo, 2*k, x, s);
        formirajDrvo(a, drvo, 2*k+1, s+1, y);
        // izračunavamo vrednost u korenu
        drvo[k] = drvo[2*k] + drvo[2*k+1];
    }
}

// najmanji stepen dvojke veci ili jednak od n
int stepenDvojke(int n) {
    int s = 1;
    while (s < n)
        s <<= 1;
    return s;
}

// na osnovu datog niza a duzine n u kom su elementi smesteni od
// pozicije 0 formira se segmentno drvo i elementi mu se smeštaju u
// niz drvo krenuvši od pozicije 1
vector<int> formirajDrvo(const vector<int>& a) {
    // niz implicitno dopunjujemo nulama tako da mu duzina postane
    // najblizi stepen dvojke
```

```

    int n = stepenDvojke(a.size());
    vector<int> drvo(n * 2);
    // krećemo formiranje od korena koji se nalazi u nizu drvo
    // na poziciji 1 i pokriva elemente na pozicijama [0, n-1]
    formirajDrvo(a, drvo, 1, 0, n - 1);
    return drvo;
}

// izračunava se zbir onih elemenata polaznog niza koji se
// nalaze na pozicijama iz segmenta [a, b] koji se nalaze u
// segmentnom drvetu koje čuva elemente polaznog niza koji se
// nalaze na pozicijama iz segmenta [x, y] i smešteno je u nizu
// drvo od pozicije k
int zbirSegmenta(const vector<int>& drvo, int k, int x, int y, int a, int b) {
    // segmenti [x, y] i [a, b] su disjunktni
    if (b < x || a > y) return 0;
    // segment [x, y] je potpuno sadržan unutar segmenta [a, b]
    if (a <= x && y <= b)
        return drvo[k];
    // segmenti [x, y] i [a, b] se seku
    int s = (x + y) / 2;
    return zbirSegmenta(drvo, 2*k, x, s, a, b) +
        zbirSegmenta(drvo, 2*k+1, s+1, y, a, b);
}

// izračunava se zbir elemenata polaznog niza dužine n koji se
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drveta
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
int zbirSegmenta(const vector<int>& drvo, int a, int b) {
    int n = drvo.size() / 2;
    // krećemo od drveta smeštenog od pozicije 1 koje
    // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
    return zbirSegmenta(drvo, 1, 0, n-1, a, b);
}

// ažurira segmentno drvo smešteno u niz drvo od pozicije k
// koje sadrži elemente polaznog niza a dužine n sa pozicija iz
// segmenta [x, y], nakon što se na poziciju i niza upiše vrednost v
void postavi(vector<int>& drvo, int k, int x, int y, int i, int v) {
    if (x == y)
        // ažuriramo vrednost u listu
        drvo[k] = v;
    else {
        // proveravamo da li se pozicija i nalazi levo ili desno
        // i u zavisnosti od toga ažuriramo odgovarajuće poddrvo
        int s = (x + y) / 2;
        if (x <= i && i <= s)
            postavi(drvo, 2*k, x, s, i, v);
        else
            postavi(drvo, 2*k+1, s+1, y, i, v);
        // pošto se promenila vrednost u nekom od dva poddrveta
        // moramo ažurirati vrednost u korenu
        drvo[k] = drvo[2*k] + drvo[2*k+1];
    }
}

// ažurira segmentno drvo smešteno u niz drvo od pozicije 1

```

```

// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void postavi(vector<int>& drvo, int i, int v) {
    int n = drvo.size() / 2;
    // krećemo od drveta smeštenog od pozicije 1 koje
    // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
    postavi(drvo, 1, 0, n-1, i, v);
}

int main() {
    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];
    vector<int> drvo = formirajDrvo(a);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        string upit;
        cin >> upit;
        if (upit == "p") {
            int i, v;
            cin >> i >> v >> ws;
            postavi(drvo, i, v);
        } else if (upit == "z") {
            int a, b;
            cin >> a >> b;
            cout << zbiriSegmenta(drvo, a, b) << '\n';
        }
    }
    return 0;
}

```

Види групачија решења овој задатка.

5.3.3 Фенвикова стабла

Фенвикова дрвешта тј. бинарно индексирана дрвешта (енгл. binary indexed tree, BIT) користе мало мање меморије и могу бити за константни фактор бржа од сегментних дрвета (иако је сложеност операција асимптотски једнака). Са друге стране, за разлику од сегментних дрвета која су погодна за различите операције, Фенвикова дрвета су специјализована само за асоцијативне операције које имају инверз (нпр. зборови или производи елемената сегмената се могу налазити уз помоћ BIT, али не и минимуми, нзд-ови и слично). Сегментна дрвета могу да ураде све што и Фенвикова, док обратно не важи.

Иако се назива дрветом, Фенвиково дрво заправо представља низ вредности зборових неких паметно изабраних сегмената. Избор сегмената је у тесној вези са бинарном репрезентацијом индекса. Поново ћемо једноставности ради претпоставити да се вредности у низу смештају од позиције 1 (вредност на позицији 0 је ирелевантна) и то и у полазном низу и у низу у ком се смешта дрво. Прилагођавање кода ситуацији у којој су у полазном низу елементи смештени од позиције нула, веома је једноставно (само је на почетку сваке функције која ради са дрветом индекс полазног низа потребно увећати за један пре даље обраде). Ако је полазни низ дужине n , елементи дрвета ће се смештати у посебан низ на позиције $[1, n]$.

Кључна идеја Фенвиковог дрвета је следећа: у дрвешћу се на позицији k чува збир вредности полазног низа из сегмената позиција облика $(f(k), k]$ где је $f(k)$ број који се добије од броја k тако што се из бинарног записа броја k обрише прва јединица десна.

На пример, на месту $k = 21$ записује се збир елемената полазног низа на позицијама из интервала $(20, 21]$, јер се број 21 бинарно записује као 10101 и брисањем јединице добија се бинарни запис 10100 тј. број 20

(важи да је $f(21) = 20$). На позицији број 20 налази се збир елемената са позиција из интервала $(16, 20]$, јер се брисањем јединице добија бинарни запис 10000 тј. број 16 (важи да је $f(20) = 16$). На позицији 16 се чува збир елемената са позиција из интервала $(0, 16]$, јер се брисањем јединице из бинарног записа броја 16 добија 0 (важи да је $f(16) = 0$).

За низ 3, 4, 1, 2, 6, 5, 1, 4, Фенвиково дрво би чувало следеће вредности.

0	1	2	3	4	5	6	7	8	k
	1	10	11	100	101	110	111	1000	k binarno
	0	0	10	0	100	100	110	0	f(k) binarno
(0,1]	(0,2]	(2,3]	(0,4]	(4,5]	(4,6]	(6,7]	(0,8]		interval
3	4	1	2	6	5	1	4		niz
3	7	1	10	6	11	1	26		drvo

Надовезивањем интервала $(0, 16]$, $(16, 20]$ и $(20, 21]$ добија се интервал $(0, 21]$ тј. префикс низа до позиције 21. Збир елемената у префиксу се, дакле, може добити као збир неколико елемената записаних у Фенвиковом дрвету. Ово, наравно, важи за произвољни индекс (не само за 21). Број елемената чијим се сабирањем добија збир префикса је само $O(\log n)$. Наиме, у сваком кораку се број јединица у бинарном запису текућег индекса смањује, а број n се записује са највише $O(\log n)$ бинарних јединица.

Имплементација је веома једноставна, када се пронађе начин да се из бинарног записа броја uklони прва јединица здесна тј. да се за дати број k израчуна $f(k)$. Под претпоставком да су бројеви записани у потпуном комплементу, изразом $k \& -k$ може се добити број који садржи само једну јединицу и то на месту последње јединице у запису броја k . Одузимањем те вредности од броја k тј. изразом $k - (k \& -k)$ добијамо ефекат брисања последње јединице у бинарном запису броја k и то представља имплементацију функције f . Други начин да се то уради је да се израчуна вредност $k \& (k-1)$.

Збир префикса $[0, k]$ полазног низа можемо онда израчунати наредном функцијом.

```
// na osnovu Fenvikovog drveta smeštenog u niz drvo
// izracunava zbir prefiksa [0, k] polaznog niza
int zbirPrefiksa(int drvo[], int k) {
    int zbir = 0;
    while (k > 0) {
        zbir += drvo[k];
        k -= k & -k;
    }
}
```

Када знамо збир префикса, збир произвољног сегмента $[a, b]$ можемо израчунати као разлику збира префикса $(0, b]$ и збира префикса $(0, a-1]$. Пошто се оба рачунају у времену $O(\log n)$, и збир сваког сегмента можемо израчунати у времену $O(\log n)$. Напоменимо и то да је због ове операције важно да асоцијативна операција која се користи у Фенвиковом дрвету има инверз (у овом случају да бисмо могли да одузимањем две вредности префикса добијамо збир произвољног сегмента).

Основна предност Фенвикових дрвета у односу на низ свих збирова префикса је то што се могу ефикасно ажурирати. Размотримо функцију која ажурира дрво након увећања елемента у полазном низу на позицији k за вредност x . Тада је за x потребно увећати све оне збирове у дрвету у којима се као сабирак јавља и елемент на позицији k . Ти бројеви се израчунавају веома слично као у претходној функцији, једино што се уместо одузимања вредности $k \& -k$ број k у сваком кораку увећава за $k \& -k$.

На пример, ако би се у претходном примеру елемент на позицији 3 увећао за вредност 4, било би потребно повећати за 4 вредности елемената Фенвиковог дрвета на позицијама 3, 4 и 8. До ових позиција бисмо дошли почев од бинарног записа броја 3 који износи 11 сабирањем са 1 (број који садржи тачно једну јединицу на позицији последње јединице у бинарном запису датог броја) чиме бисмо добили 100 што одговара броју 4, а након тога бисмо ову вредност сабрали са 100 чиме бисмо добили 1000 (бинарни запис броја 8). Овде се процедура завршава с обзиром на то да смо стигли до последњег елемента у Фенвиковом дрвету.

```
// Ažurira Fenvikovo drvo smešteno u niz drvo nakon što se
// u originalnom nizu element na poziciji k uveća za x
void dodaj(int drvo[], int n, int k, int x) {
    while (k <= n) {
        drvo[k] += x;
    }
}
```

```

    k += k & -k;
}
}

```

Објаснимо и докажимо коректност претходне имплементације. Потребно је ажурирати све оне позиције m чији придружени сегмент садржи вредност k , тј. све оне позиције m такве да је $k \in (f(m), m]$, тј. $f(m) < k \leq m$. Ово никако не може да важи за бројеве $m < k$, а сигурно важи за број $m = k$, јер је $f(k) < k$, када је $k > 0$ (а ми претпостављамо да је $1 \leq k \leq n$). За бројеве $m > k$, сигурно важи десна неједнакост и потребно је утврдити да важи лева. Нека је $g(k)$ број који се добија од k тако што се k сабере са бројем који има само једну јединицу у свом бинарном запису и то на позицији на којој се налази последња јединица у бинарном запису броја k . На пример, за број $k = 101100$, број $g(k) = 101100 + 100 = 110000$. У имплементацији се број $g(k)$ лако може израчунати као $k + (k \& -k)$. Тврдимо да је најмањи број m који задовољава услов $f(m) < k < m$ управо $g(k)$. Заиста, очигледно важи $k < g(k)$ и $g(k)$ има све нуле од позиције последње јединице у бинарном запису броја k (укључујући и њу), па до краја, па се брисањем његове последње јединице, тј. израчунавањем $f(g(k))$ сигурно добија број који је строго мањи од k . Ниједан број m између k и $g(k)$ не може да задовољи услов да је $f(m) < k$. Наиме, сви ти бројеви се поклапају са бројем k на свим позицијама пре крајњих нула, а на позицијама крајњих нула броја k имају бар неку јединицу, чијим се брисањем добија број који је већи или једнак k . По истом принципу закључујемо да наредни тражени број мора бити $g(g(k))$, затим $g(g(g(k)))$ итд. све док се не добије неки број који превазилази n . Заиста, важи да је $k < g(k) < g(g(k))$. Важи да је $f(g(g(k))) < f(g(k)) < k$, па $g(g(k))$ задовољава услов. Ниједан број између $g(k)$ и $g(g(k))$ не може да задовољи услов, јер се сви они поклапају са $g(k)$ у свим бинарним цифрама, осим на његовим крајњим нулама где имају неке јединице. Брисањем последње јединице се добија број који је већи или једнак $g(k)$, па добијени број не може бити мањи од k . Отуда следи да су једине позиције које треба ажурирати управо позиције из серије $k, g(k), g(g(k))$ итд., све док су оне мање или једнаке n , па је наша имплементација коректна.

Остаје још питање како у старту формирати Фенвиково дрво, међутим, формирање се може свести на то да се креира дрво попуњено само нулама, а да се затим увећава вредност једног по једног елемента низа претходном функцијом.

```

// Na osnovu niza a u kom su elementi smešteni
// na pozicijama iz segmenta [1, n] formira Fenvikovo drvo
// i smešta ga u niz drvo (na pozicije iz segmenta [1, n])
void formirajDrvo(int drvo[], int n, int a[]) {
    fill_n(a+1, n, 0);
    for (int k = 1; k <= n; k++)
        dodaj(drvo, n, k, a[k]);
}

```

Задатак: Суме сегмената променљивог низа

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види шекст задатак.*

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

Решење

Задатак је могуће решити коришћењем Фенвиковог дрвета.

```

#include <iostream>
#include <vector>

using namespace std;

// na osnovu Fenikovog drveta smestenog u niz drvo
// izracunava zbir prefiksa [0, k) polaznog niza
int zbirPrefiksa(const vector<int>& drvo, int k) {
    int zbir = 0;
    while (k > 0) {
        zbir += drvo[k];
        k -= k & -k;
    }
}

```

```
    return zbir;
}

// na osnovu Fenvikovog drveta smestenog u niz drvo
// racuna zbir segmenta [a, b] polaznog niza
int zbirSegmenta(const vector<int>& drvo, int a, int b) {
    return zbirPrefiksa(drvo, b+1) - zbirPrefiksa(drvo, a);
}

// vraca element originalnog niza na poziciji a
int element(const vector<int>& drvo, int a) {
    return zbirSegmenta(drvo, a, a);
}

// azurira Fenvikovo drvo smesteno u niz drvo nakon sto se
// u originalnom nizu element na poziciji a uveca za x
void dodaj(vector<int>& drvo, int a, int x) {
    int k = a + 1;
    int n = drvo.size();
    while (k < n) {
        drvo[k] += x;
        k += k & -k;
    }
}

// azurira Fenvikovo drvo smesteno u niz drvo nakon sto se
// u originalnom nizu element na poziciji a postavi na x
void postavi(vector<int>& drvo, int a, int x) {
    dodaj(drvo, a, x - element(drvo, a));
}

// na osnovu niza a u kom su elementi smesteni na pozicijama [0, n)
// formira Fenvikovo drvo i smesta ga u niz drvo
// (na pozicije iz segmenta [1, n])
vector<int> formirajDrvo(const vector<int>& a) {
    int n = a.size();
    vector<int> drvo(n + 1, 0);
    for (int k = 0; k < n; k++)
        dodaj(drvo, k, a[k]);
    return drvo;
}

int main() {
    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];
    vector<int> drvo = formirajDrvo(a);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        string upit;
        cin >> upit;
        if (upit == "p") {
            int i, v;
            cin >> i >> v >> ws;
```

```

    postavi(drvo, i, v);
} else if (upit == "z") {
    int a, b;
    cin >> a >> b;
    cout << zbirSegmenta(drvo, a, b) << '\n';
}
}
return 0;
}

```

Задатак: Увећања сегмената и читање елемената

Напиши програм који омогућава две врсте упита над низом целих бројева: увећавање елемената датог сегмента за неку дату вредност и читавање појединачних елемената низа. Почетна вредност свих елемената низа је 0.

Улаз: Са стандардног улаза се учитава димензија низа n ($1 \leq n \leq 50000$), затим из наредне линије број упита q и након тога у наредних q линија описи упита. Упити су описани на следећи начин:

- $u\ a\ b\ k$ – након извршавања овог упита сви елементи низа на позицијама из интервала $[a, b]$ треба да буду увећани за вредност k (важи $0 \leq a \leq b < n$ и $-10 \leq k \leq 10$).
- $e\ i$ – извршавање овог упита подразумева да се на стандардни излаз испише вредност елемента низа на позицији i (важи $0 \leq i < n$).

Излаз: На стандардни излаз исписати вредности свих e упита (сваки број у посебној линији).

Пример

Улаз	Излаз
10	5
6	3
u 3 5 2	-2
u 4 8 3	
e 4	
e 7	
u 1 7 -5	
e 6	

Објашњење

Након првог упита стање низа је 0 0 0 2 2 2 0 0 0 0, након другог 0 0 0 2 5 5 3 3 3 0. Тада се читавају елементи на позицијама 4 и 7. Након трећег упита стање низа је 0 -5 -5 -3 0 0 -2 -2 3 0. Након тога се читава елемент на позицији 6.

Решење

Директно решење

Задатак можемо решити директно одржавајући низ елемената. Сложеност операције увећања елемената је $O(n)$, а читавања вредности је $O(1)$. Укупна сложеност је, дакле, $O(qn)$.

```

#include <iostream>
#include <vector>

using namespace std;

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);
    int n; cin >> n;
    vector<int> niz(n);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        string upit;

```

```

cin >> upit;
if (upit == "u") {
    int a, b, v;
    cin >> a >> b >> v;
    for (int i = a; i <= b; i++)
        niz[i] += v;
} else if (upit == "e") {
    int a;
    cin >> a;
    cout << niz[a] << '\n';
}
}
}

```

Фенвиково дрво низа разлика

У задатку **Увећавање сегмената** видели смо да се увећавање сегмената може урадити у времену $O(1)$, ако се уместо оригиналног низа одржава низ разлика суседних елемената. Међутим, тада операција читања елемената захтева израчунавање збира префикса низа разлика, то је операција сложености $O(n)$. Ако уместо класичног низа, низ разлика одржавамо у Фенвиковом дрвету, тада и ажурирање елемената и сабирање префикса вршимо у времену $O(\log n)$, што је добар компромис. Увећавање сегмената оригиналног низа ће се свести на два увећавања елемената низа разлика, што су две операције над Фенвиковим дрветом сложености $O(\log n)$. Читање елемената оригиналног низа ће се свести на израчунавање збира префикса низа разлика, што је једна операција над Фенвиковим дрветом сложености $O(\log n)$. Укупна сложеност ће бити $O(q \log n)$.

```

#include <iostream>
#include <vector>

```

```

using namespace std;

```

```

// operacije za rad sa Fenvikovim drvetom

```

```

// azurira Fenvikovo drvo smesteno u niz drvo nakon sto se
// u originalnom nizu element na poziciji k uveca za x

```

```

void dodaj(vector<int>& drvo, int k, int x) {
    k++;
    int n = drvo.size();
    while (k < n) {
        drvo[k] += x;
        k += k & -k;
    }
}

```

```

// na osnovu Fenvikovog drveta smestenog u niz drvo
// izracunava zbir prefiksa [0, k) originalnog niza

```

```

int zbirPrefiksa(const vector<int>& drvo, int k) {
    int zbir = 0;
    while (k > 0) {
        zbir += drvo[k];
        k -= k & -k;
    }
    return zbir;
}

```

```

void uvecajSegment(vector<int>& drvo, int a, int b, int v) {
    dodaj(drvo, b+1, -v);
    dodaj(drvo, a, v);
}

```

```

int element(const vector<int>& drvo, int a) {

```



```

    return zbirPrefiksa(drvo, a+1);
}

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);
    int n; cin >> n;
    vector<int> drvo(n+1);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        string upit;
        cin >> upit;
        if (upit == "u") {
            int a, b, v;
            cin >> a >> b >> v;
            uvecajSegment(drvo, a, b, v);
        } else if (upit == "e") {
            int a;
            cin >> a;
            cout << element(drvo, a) << '\n';
        }
    }
    return 0;
}

```

Задатак: К-ти парни број

Написати програм који омогућава да се у низу природних бројева који је на почетку испуњен нулама, али чији се елементи често мењају током извршавања програма ефикасно проналази позиција k -тог парног броја по реду.

Улаз: Са стандардног улаза се учитава дужина низа n ($1 \leq n \leq 50000$), а затим и број упита m ($1 \leq m \leq 50000$). Извршавањем упита облика $u\ p\ x$ се у низ на позицију $1 \leq p \leq n$ уписује број x , док се упитом $c\ k$ на стандардни излаз исписује позиција k -тог парног броја у текућем садржају низа (позиције се броје од 1).

Излаз: На стандардном излазу приказати резултате извршавања упита c . Ако у неком случају у низу има мање парних бројева од вредности k , тада уместо позиције исписати $-$.

Пример

Улаз	Излаз
5	5
8	4
u 3 1	4
c 4	-
u 1 7	
u 2 5	
c 1	
u 1 2	
c 2	
c 4	

Решење

Директно решење подразумева да се елементи уписују у низ и да се позиција k -тог парног одређује применом линеарне претраге. Ако имамо m_1 операција ажурирања и m_2 операција претраге сложеност таквог решења је $O(m_1 + m_2 \cdot n)$, што може бити прилично неефикасно.

```

#include <iostream>
#include <vector>

```

```

using namespace std;

```

```

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);
    // elementi niza - dodajemo jedan element vise, jer se pozicije broje od 1
    int n;
    cin >> n;
    vector<int> niz(n+1, 0);
    // broj upita
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        char c;
        cin >> c;
        if (c == 'u') {
            // upit upisa elementa u niz
            int p, x;
            cin >> p >> x;
            niz[p] = x;
        } else if (c == 'c') {
            // upit odredjivanja k-tog parnog broja
            int k;
            cin >> k;
            // linearnom pretragom odredjujemo poziciju k-tog parnog broja
            int brojParnih = 0;
            for (int p = 1; p <= n; p++)
                if (niz[p] % 2 == 0) {
                    brojParnih++;
                    if (brojParnih == k) {
                        cout << p << '\n';
                        break;
                    }
                }
            // ako nema k parnih brojeva, ispisujemo -
            if (brojParnih < k)
                cout << "- " << '\n';
        }
    }
    return 0;
}

```

На основу низа можемо формирати низ нула и јединица такав да се јединице налазе на месту парних елемената. Тада је позиција k -тог по реду парног броја најмања позиција таква да је збир свих јединица закључно са том позицијом једнак k . Ако низ нула и јединица одржавамо у Фенвиковом стаблу, врло ефикасно можемо да израчунавамо збирове сваког фиксираног префикса. Захваљујући чињеници да су зборови префикса монотонно неоппадајући (када се префикси продужавају) тражену позицију можемо ефикасно одредити алгоритмом бинарне претраге. Ако имамо m_1 операција ажурирања и m_2 операција претраге укупна сложеност ће бити $O(m_1 \log(n) + m_2 \log^2 n)$, тј. $O(m \log^2 n)$.

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
// operacije za rad sa Fenwickim drvetom
```

```

void dodaj(vector<int>& drvo, int k, int v) {
    while (k < (int)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;
}

// vraca prvu poziciju p tako da je zbir niza na pozicijama [1, p]
// veci ili jednak k
int prefiksK(const vector<int>& drvo, int k) {
    // poziciju pronalazimo binarnom pretragom po vrednosti zbira prefiksa
    int l = 1, d = drvo.size() - 1;
    while (l <= d) {
        int s = l + (d - l) / 2;
        if (zbirPrefiksa(drvo, s) < k)
            l = s + 1;
        else
            d = s - 1;
    }
    return l;
}

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);
    int n;
    cin >> n;
    // gradimo Fenvikovo drvo i inicijalizujemo ga jedinicama
    vector<int> drvo(n + 1, 0);
    for (int k = 1; k <= n; k++)
        dodaj(drvo, k, 1);
    // elementi niza
    vector<int> niz(n + 1, 0);

    // broj upita
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        char c;
        cin >> c;
        if (c == 'u') {
            // upit upisa elementa u niz
            int p, x;
            cin >> p >> x;
            if (x % 2 == 0 && niz[p] % 2 != 0)
                // upisan je novi paran element na poziciju p
                dodaj(drvo, p, 1);
            else if (x % 2 != 0 && niz[p] % 2 == 0)
                // upisan je novi neparan element na poziciju p
                dodaj(drvo, p, -1);
            niz[p] = x;
        } else if (c == 'c') {
            // upit odredjivanja k-tog parnog elementa
            int k;
            cin >> k;

```

```

// trazimo najkraci prefiks ciji je zbir jednak k
int p = prefiksK(drvo, k);
// proveravamo da li takav prefiks zaista postoji
if (p <= n)
    cout << p << "\n";
else
    cout << "- " << "\n";
}
}
return 0;
}

```

Задатак: Трик са картама

Мали мађионичар хоће да сложи карте тако да са њима изведе следећи трик. У првом кораку једну карту са врха шпила пребацује на дно шпила, узима следећу карту са врха шпила и испоставља се да је на њој број 1. У другом кораку пребацује два пута карту са врха шпила на дно шпила, узима следећу карту са врха шпила испоставља се да је на њој број два. Наставља да ради по истом принципу, пребацујући у сваком кораку по једну карту више са врха на дно шпила, да би се након тога са врха шпила скинула управо карта са наредним редним бројем.

Улаз: Са стандардног улаза се учитава број карата у шпилу n ($1 \leq n \leq 50000$).

Изаз: На стандардни излаз исписати редослед карата у шпилу, од врха ка дну (након сваке карте исписати један размак).

Пример

Улаз	Изаз
12	7 1 4 9 2 11 10 8 3 6 5 12

Решење

Ако кренемо од празног низа (можемо га иницијално попунити нулама), карте можемо попунити тако што кренемо од почетка, затим прескочимо једно празно место и упишемо јединицу, затим прескочимо два празна места и упишемо двојку, затим прескочимо три празна места и упишемо тројку и тако даље. Можемо приметити да када се број повећа и број празних места смањи, прескакање празних места и по неколико пута обилази низ. Ако прескочимо сва празна места у низу (којих у k -том кораку има $n - k + 1$ долазимо на исто место са ког смо пошли, па се убрзање може постићи тако што пре прескакања k празних места пронађемо остатак тог броја при дељењу са $n - k + 1$.

```

#include <iostream>
#include <vector>

```

```
using namespace std;
```

```

int main() {
    int n;
    cin >> n;
    vector<int> karta(n, 0);
    // trenutna pozicija
    int p = n-1;
    for (int k = 1; k <= n; k++) {
        // treba da preskocimo k praznih pozicija
        // posto je preostalo n-k+1 praznih, one se ciklicno ponavljaju,
        // pa je dovoljno preskociti k % (n-k+1) praznih pozicija
        for (int i = 1; i <= (k % (n-k+1)) + 1; i++) {
            // pomeramo se na narednu poziciju i preskacemo sve
            // pune pozicije do naredne prazne
            while (karta[p = (p + 1) % n] != 0)
                ;
        }
        // upisujemo broj k na trenutnu poziciju
    }
}

```

```

    karta[p] = k;
}
// ispisujemo konacan rezultat
for (int k : karta)
    cout << k << " ";
cout << endl;
return 0;
}

#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> karta(n, 0);
    // trenutna pozicija
    int p = n-1;
    for (int k = 1; k <= n; k++) {
        // treba da preskocimo k praznih pozicija
        // posto je preostalo n-k+1 praznih, one se ciklicno ponavljaju,
        // pa je dovoljno preskociti k % (n-k+1) praznih pozicija
        for (int i = 1; i <= (k % (n-k+1)) + 1; i++) {
            // pomeramo se na narednu poziciju i preskacemo sve
            // pune pozicije do naredne prazne
            while (karta[p = (p + 1) % n] != 0)
                ;
        }
        // upisujemo broj k na trenutnu poziciju
        karta[p] = k;
    }
    // ispisujemo konacan rezultat
    for (int k : karta)
        cout << k << " ";
    cout << endl;
    return 0;
}

```

Празна места (тј. њихове позиције) можемо чувати у листи, јер нам она даје могућност да веома ефикасно избацимо елемент из средине. Приликом кретања по листи морамо водити рачуна о томе да када стигнемо до краја, обилазак кренемо из почетка.

Међутим, још ефикасније решење од употребе листе је то да слободну позицију која следи након прескакања k слободних позиција одредимо као m -ту слободну позицију у низу. Наиме, ако знамо да пре текуће позиције постоји p слободних позиција тада уместо да од текуће позиције прескачемо p позиција можемо од почетка низа прескочити $p+k$ позиција и елемент k уписати на позицију $p+k+1$. На пример, ако у низу $. 1 . . 2 . . .$ тражимо позицију на коју треба да упишемо број 3 од елемента 2 треба да прескочимо 3 празне позиције, међутим, пошто пре елемента 2 постоји 3 празне позиције од почетка низа прескачемо $3 + 3 = 6$ празних позиција и елемент 3 уписујемо на седму празну позицију у низу чиме добијамо $. 1 . . 2 . . . 3 ..$ Ичак, треба бити обазрив, јер се некада долази до краја низа, па се бројање враћа на почетак низа. Зато број празне позиције на коју треба уписати елемент k заправо није увек $p+k+1$, већ је једнак $\text{mod } (p+k)(n-k+1)+1$. Нпр. ако у низу $. 1 . . 2 . . . 3 .$ желимо да одредимо позицију елемента 4 знамо да се пре елемента 3 налази 6 празних позиција, што значи да елемент 4 треба да упишемо на празну позицију чији је редни број $\text{mod } (6+4)7+1 = 4$, чиме добијамо низ $. 1 . . 2 4 . . 3 ..$ Слично, пошто пре елемента 4 има 3 празне позиције елемент 5 уписујемо на позицију са редним бројем $\text{mod } (3+5)6+1 = 3$ чиме добијамо низ $. 1 . 5 2 4 . . 3 ..$

Остаје питање како одредити k -ту празну позицију у низу, међутим, то можемо урадити истом техником којом

5.3. УПИТИ РАСПОНА

смо у задатку **K -ти парни број** одређивали k -ти по реду паран број у низу који се мења (овде је ажурирање само такво да празне позиције могу постати непразне).

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

using namespace std;

// операције за рад са Фенвиковим дрветом

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;
}

// враћа прву позицију p тако да је збир нiza на позициjама [1, p]
// већи или једнак k
int prefiksK(const vector<int>& drvo, int k) {
    // позицију проналазимо бинарном претрагом по вредности збира префикса
    int l = 1, d = drvo.size() - 1;
    while (l <= d) {
        int s = l + (d - l) / 2;
        if (zbirPrefiksa(drvo, s) < k)
            l = s + 1;
        else
            d = s - 1;
    }
    return l;
}

int main() {
    // број и низ карата
    int n;
    cin >> n;
    vector<int> karta(n, 0);

    // Фенвиково дрво над низом који чува 1 на празним и 0 на
    // заузетим позициjама (бројање креће од 1)
    vector<int> drvo(n + 1, 0);
    for (int k = 1; k <= n; k++)
        dodaj(drvo, k, 1);

    // тренутна слободна позиција
    int p = 0;
    for (int k = 1; k <= n; k++) {
        // треба да се померимо за k празних позиција
```

```

// posto je preostalo n-k+1 praznih, one se ciklicno ponavljaju,
// pa je dovoljno preskociti k % (n-k+1) praznih pozicija

// odredjujemo najmanju poziciju takvu da se zakljucno sa njom
// nalazi za k slobodnih pozicija vise nego sto se nalazi
// zakljucno sa trenutnom slobodnom pozicijom (sve po modulu n-k+1)
p = prefiksK(drvo, (zbirPrefiksa(drvo, p) + k) % (n-k+1) + 1);

// upisujemo kartu, poziciju oznacavamo punom i azuriramo drvo drvo
karta[p-1] = k;
dodaj(drvo, p, -1);
}

// ispisujemo konacan raspored karata
for (int k : karta)
    cout << k << " ";
cout << endl;
return 0;
}

```

Задатак: Инверзије након избацивања сегмената

Дат је низ a , који се састоји од n позитивних целих бројева. Написати програм који ће да преброји колико има парова бројева l и r таквих да је $1 \leq l < r \leq n$ и да низ $b = a_1 a_2 \dots a_l a_r a_{r+1} \dots a_n$ нема више од k инверзија.

Два цела броја у низу b образују инверзију када се већи број појављује пре мањег броја, тј. за пар b_i, b_j чланова низа b кажемо да образују инверзију ако важи да је $1 \leq i < j \leq |b|$ и $b_i > b_j$, где је $|b|$ величина низа b .

Улаз: У првој линији стандардног улаза дата су два цела броја k и n ($2 \leq n \leq 10^5, 0 \leq k \leq 10^{18}$) – максималан број дозвољених инверзија и величина низа a . Следећа линија садржи n позитивних целих бројева раздвојених са по једним размаком $a_1, a_2, \dots, a_n$ ($1 \leq a_i \leq 10^9$) – елементи низа a .

Ишлаз: У јединој линији стандардног излаза исписати само један број – број парова (решење задатка).

Пример 1		Пример 2		Пример 3		Пример 4	
Улаз	Ишлаз	Улаз	Ишлаз	Улаз	Ишлаз	Улаз	Ишлаз
1	3	0	1	2	6	4	10
3		3		5		5	
1 3 2		1 3 2		1 5 4 1 100		1 5 4 1 100	

Решење**Груба сила**

Решење грубом силом подразумева да се обраде сви парови (l, r) и да се за сваки пар преброје инверзије низа. Пошто парова има $O(n^2)$, ово решење је веома неефикасно, чак и када се инверзије броје на неки ефикасан начин (као што је приказано у задатку [Број инверзија](#)). Наиме, број инверзија датог низа се може грубом силом одредити у сложености $O(n^2)$, што доводи до укупне сложености од чак $O(n^4)$. Ефикаснијим алгоритмима се број инверзија може одредити у сложености $O(n \log n)$, што доводи до укупне сложености $O(n^3 \log n)$ што је и даље недопустиво велико.

```

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

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

    int n;
    cin >> n;

```

```

vector<int> a(n);
for (int i = 0; i < n; i++)
    cin >> a[i];
int brojParova = 0;
for (int l = 0; l < n-1; l++) {
    for (int r = l + 1; r < n; r++) {
        int brojInverzija = 0;
        for (int i = 0; i < n && brojInverzija <= maksInverzija; i++)
            if (i <= l || i >= r)
                for (int j = i+1; j < n && brojInverzija <= maksInverzija; j++)
                    if (j <= l || j >= r)
                        if (a[i] > a[j])
                            brojInverzija++;
        if (brojInverzija <= maksInverzija)
            brojParova++;
    }
}
cout << brojParova << endl;
return 0;
}

```

Техника два показивача

У основи ефикаснијег решења лежи техника два показивача која нам омогућава да експлицитно проверимо само линеарни број парова (l, r) .

Означимо са $I(l, r)$ број инверзија низа одређеног паром (l, r) (тј. број инверзија низа $a_0, \dots, a_l, a_r, \dots, a_{n-1}$ и са I_{max} максимални допуштени број инверзија.

Желимо да за сваки десни крај r одредимо број левих крајева l таквих да је $I(l, r) \leq I_{max}$.

Ако пронађемо највећу вредност $l_{max} < r$ такву да је $I(l_{max}, r) \leq I_{max}$, знамо да ће важити и да је $I(0, r) \leq I_{max}$, $I(1, r) \leq I_{max}$, ..., $I(l_{max}, r) \leq I_{max}$ (јер за мање вредности левог краја број искључених елемената низа расте, па инверзија може бити само мање него у низу одређеном паром (l_{max}, r)), док ће са друге стране важити $I(l_{max} + 1, r) > I_{max}$, ..., $I(r - 1, r) > I_{max}$ (јер за веће вредности левог краја број искључених елемената низа опада, па инверзија може бити само више него у низу одређеном паром (l_{max}, r)). То значи да се налажењем те највеће вредности l_{max} може утврдити да постоји тачно $l_{max} + 1$ један пар са десним крајем r који одређује низ са допуштеним бројем инверзија.

Још један случај који се може јавити је да ни за једно l не важи да је да је број инверзија низа одређеног паром (l, r) унутар допуштеног прага тј. да је $I(0, r) > I_{max}$ и тада смо сигурни да не постоји ни један пар одређен десним крајем r који одређује низ са допуштеним бројем инверзија, па ни вредност l_{max} није коректно дефинисана.

Зато свако r дефинишимо функцију која одређује тражену вредност l , тј. нека је $l(r) = l_{max}$ ако таква вредност постоји, односно да је $l(r) = 0$ ако таква вредност не постоји и тражени број парова једнак је збиру вредности $l(r) + 1$ за оне r за које је $l(r) = l_{max}$ тј. за које је $I(0, r) \leq I_{max}$ (оне вредности r за које је $I(0, r) > I_{max}$ не доприносе броју парова).

Имплементацију организујемо око петље у којој се одржавају две променљиве $l < r$ и променљива I у којој чувамо број инверзија $I(l, r)$. Вредности иницијализујемо на $l = 0$ и $r = 1$, док број инверзија иницијализујемо на укупан број инверзија у низу (о његовом ефикасном израчунавању биће речи мало касније).

За свако r покушавамо да пронађемо $l(r)$. Ако $I = I(l, r) \leq I_{max}$ и ако је $l+1 < r$, тада је $l(r) \geq l$. Увећањем вредности l у низ који се разматра се додаје елемент a_{l+1} , чиме се број инверзија увећава тачно за број инверзија у којима учествује тај елемент. Тај број је одређен елементима у интервалу $[0, l]$ који су строго већи од a_{l+1} и елементима у интервалу $[r, n)$ који су строго мањи од a_{l+1} (и о начину њиховог ефикасног проналажења ће бити мало више речи касније). Ако је збир текућег броја инверзија I и броја инверзија одређеног елементом a_{l+1} мањи или једнак од допуштеног прага I_{max} се увећава и поступак се понавља све док l не достигне вредност $r - 1$ или док се не пронађе вредност l таква да се укључивањем елемента $l + 1$ број инверзија повећава изнад прага (то је тражена вредност $l(r) = l_{max}$). На тај начин одређујемо нашу тражену последњу вредност l и број пронађених парова повећавамо за $l + 1$. Ако је $l = 0$ и ако је $I(l, r) > I_{max}$, тада је $l(r) = 0$.

Након проналажења вредности $l(r)$ за тренутну вредност r , можемо прећи на наредну вредност десног краја тј. на вредност $r + 1$. Потребно је да одредимо $l(r + 1)$. Повећањем вредности r из низа избацујемо вредност a_r и број инверзија ажурирамо тако што одузмемо тачно оне инверзије у којима је учествовао елемент a_r . Тај број је одређен свим елементима из интервала $[0, l]$ који су строго већи од a_r и свим елементима из интервала $[r + 1, n)$ који су строго мањи од a_r (и о њиховом ефикасном одређивању биће више речи касније). Након тога се увећава r и потребно је одредити $l(r)$ за увећано r . Кључна опаска је да l не морамо тражити из почетка, већ претрагу можемо наставити од тренутне вредности l , јер је $l(r)$ моното неопадајућа функција по r . Наиме, ако је $l(r) = 0$ и ако је $I(0, r) > I_{max}$, тада је тривијално $l(r + 1) \geq 0$. У супротном, ако је $l(r) = l_{max}$ тада је $I(l, r) \leq I_{max}$, па је $I(l, r + 1) \leq I(l, r)$ (повећавањем вредности r број инверзија се само може смањити), па је онда $l(r + 1) \geq l_{max}$ (важи да је $I(l_{max}, r) \leq I_{max}$, па се последња позиција l_{max} за увећано r може налазити само десно од последње позиције $l = l_{max}$ пре увећања r). Зато након увећања r , претрагу за $l(r)$ настављамо од текуће вредности l , тако што l увећавамо све док не достигне вредност $r - 1$ или вредност такву да је $I(l, r) \leq I_{max}$ и $I(l + 1, r) > I_{max}$ (то је онда вредност l_{max}).

С обзиром на то да се и l и r само могу увећавати, највећи број итерација ове петље је $2n$.

Остаје још неколико питања које је потребно решити да би се добила ефикасна имплементација:

- како одредити почетни број инверзија,
- како одређивати број инверзија за елемент a_{l+1} који се убацује у низ и
- како одређивати број инверзија за елемент a_r који се избацује из низа?

Један начин је да се ово уради грубом силом. Почетни број инверзија се може грубом силом одредити у сложености $O(n^2)$ и то се ради само једном. Број инверзија са сваки конкретан елемент се може грубом силом урадити у сложености $O(n)$. Тај корак се извршава $O(n)$ пута (приликом сваке промене вредности l и r). Све заједно сложеност је $O(n^2)$.

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

long long brojInverzija(const vector<int>& a) {
    int broj = 0;
    for (int i = 0; i < a.size(); i++)
        for (int j = i+1; j < a.size(); j++)
            if (a[i] > a[j])
                broj++;
    return broj;
}

int levoVecih(const vector<int>& a, int l, int x) {
    int broj = 0;
    for (int i = 0; i <= l; i++)
        if (a[i] > x)
            broj++;
    return broj;
}

int desnoManjih(const vector<int>& a, int r, int x) {
    int broj = 0;
    for (int i = r; i < a.size(); i++)
        if (a[i] < x)
            broj++;
    return broj;
}

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

```

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

long long brojParova = 0;
int l = 0, r = 1;
long long inverzija = brojInverzija(a);
while (r < a.size()) {
    while (true) {
        if (l+1 == r)
            break;
        int uvecanje = levoVecih(a, l, a[l+1]) + desnoManjih(a, r, a[l+1]);
        if (inverzija + uvecanje > maksInverzija)
            break;
        l++;
        inverzija += uvecanje;
    }
    if (inverzija <= maksInverzija)
        brojParova += l + 1;
    int umanjenje = levoVecih(a, l, a[r]) + desnoManjih(a, r+1, a[r]);
    inverzija -= umanjenje;
    r++;
}
cout << brojParova << endl;

return 0;
}

```

Структуре података

За ефикасно одређивање укупног броја инверзија и броја инверзија датог елемента можемо употребити ефикасне структуре података. У сва три случаја потребно је да имамо структуру података која подржава следеће операције: додавање елемента у скуп, избацивање елемента из скупа одређивање броја елемената скупа који су мањи од дате вредности и одређивање броја елемената скупа који су већи од дате вредности. Пошто у низу може бити једнаких елемената, уместо скупа је потребно употребити неки облик мултискупа.

Мултискупови

Делује наизглед да добро решење може бити библиотечка колекција `multiset`. Уметање и брисање је могуће извршити у времену $O(\log n)$. Број елемената строго већих од датог можемо одредити тако што пронађемо итератор на први елемент који је строго већи од датог (методом `upper_bound`) и одредимо његово растојање од краја низа (функцијом `distance`). Слично, број елемената који су строго мањи од датог можемо одредити тако што пронађемо итератор на први елемент који је већи или једнак од датог (методом `lower_bound`) и пронађемо растојање од почетка низа до тог елемента (функцијом `distance`). Методе `lower_bound` и `upper_bound` су ефикасне (захтевају логаритамско време у односу на број елемената скупа), међутим, растојање између два итератора колекције `multiset` тј. функција `distance` захтева линеарно време (за разлику од итератора у линеарним колекцијама попут вектора, где је то време константно). Због тога је укупна временска сложеност овог приступа квадратна (линеарни број пута се померају границе l и r и при сваком померању морамо да урадимо операције над скупом које су линеарне сложености).

```

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
using namespace std;

// број елемената у скупу који су строго већи од вредности k
long long brojVecihOd(int k, const multiset<int>& skup) {
    return distance(skup.upper_bound(k), skup.end());
}

```

```

// broj elemenata u skupu koji su strogo manji od vrednosti k
long long brojManjihOd(int k, const multiset<int>& skup) {
    return distance(skup.begin(), skup.lower_bound(k));
}

int main() {
    // učitavamo ulazne podatke
    long long maksInverzija;
    cin >> maksInverzija;
    int n;
    cin >> n;
    vector<int> a(n);
    for(int i = 0; i < n; i++)
        cin >> a[i];

    // brojimo ukupan broj inverzija polaznog niza
    long long brojInverzija = 0;
    multiset<int> vidjeni;
    for (int i = 0; i < n; i++) {
        brojInverzija += brojVecihOd(a[i], vidjeni);
        vidjeni.insert(a[i]);
    }

    // trazeći broj parova (l, r)
    long long brojParova = 0;

    // krećemo od para (0, 1)
    int l = 0, r = 1;

    // skup elemenata na pozicijama [0, L]
    multiset<int> DoL;
    DoL.insert(a[0]);
    // skup elemenata na pozicijama [R, n)
    multiset<int> OdR = vidjeni;
    OdR.erase(OdR.find(a[0]));

    // za svaku moguću vrednost desnog kraja
    while (r < a.size()) {
        // određujemo poslednju vrednost l takvu da je broj inverzija kada se
        // izbace elementi iz intervala (l, r) manji od zadate granice
        while (true) {
            // posto mora da vazi l < r, l više nije moguće uvećati
            if (l+1 == r)
                break;
            // ispitujemo da li je moguće povećati l, tj. ubaciti element a[l+1]
            int uvecanje = brojVecihOd(a[l+1], DoL) + brojManjihOd(a[l+1], OdR);
            // ako nije, trenutno l je poslednje moguće
            if (brojInverzija + uvecanje > maksInverzija)
                break;
            // u suprotnom ubacujemo a[l+1] u skup, azuירamo broj inverzija
            // i povećavamo l
            brojInverzija += uvecanje;
            DoL.insert(a[l+1]);
            l++;
        }
    }
}

```

```

// ako je broj inverzija za trenutni par (l, r) ispod dopustene
// granice, takav je i broj inverzija za sve parove
// (0, r), (1, r), ..., (l, r)
if (brojInverzija <= maksInverzija)
    brojParova += l + 1;

// uvecavamo r tako sto izbacujemo a[r] iz skupa i azuriramo broj
// inverzija
int umanjenje = brojVecihOd(a[r], DoL) + brojManjihOd(a[r], OdR);
brojInverzija -= umanjenje;
OdR.erase(OdR.find(a[r]));
r++;
}

// ispisujemo konacan rezultat
cout << brojParova << endl;

return 0;
}

```

Фенвиково дрво

Ако знамо да елементи скупа долазе из неког интервала облика $[1, n]$, мултикуп можемо реализовати помоћу низа у ком на свакој позицији i чувамо број појављивања елемента i . Пошто не знамо унапред распон вредности, на почетку рада елементе низа преводимо у њихов ранг тј. редослед у сортираном низу, при чему ранг бројимо од 1. Та операција не утиче на број инверзија.

Додавање елемента у скуп се своди на увећање броја његових појављивања за 1, а уклањање елемента из скупа на умањивање броја његових појављивања за 1. Број елемената мањих од датог елемента i одређујемо као збир префикса низа на позицијама $[1, i)$, а број елемената већих од датог елемента i као збир суфикса низа на позицијама $[i, n)$. Дакле, потребна нам је структура података која нам допушта да у низу чији се елементи често мењају ефикасно израчунавамо збирове произвољних префикса и суфикса.

Једно ефикасно решење можемо добити ако употребимо *Фенвиково дрво* (тј. *бинарно-индексирано дрво*, BIT). Подсетимо се, оно у листовима чува вредности неког низа, док у унутрашњим чворовима чува парцијалне збирове одређених сегмената који нам онда омогућавају да у логаритамској сложености одредимо збир произвољног префикса дрвета. Пошто се збир било ког сегмента (укључујући и суфиксе) може добити као разлика збира два префикса, Фенвиково дрво нам омогућава да ефикасно одредимо збир било ког сегмента оригиналног низа.

Овим сложеност операције додавања елемента у мултикуп, брисања елемента из мултикупа, као и одређивања броја елемената мањих од датог и одређивања броја елемената већих од датог постаје логаритамска у односу на број елемената у мултикупу. Зато је одређивање броја инверзија сваког појединачног елемента сложености $O(\log n)$, па одређивање почетног броја инверзија има сложеност $n \log n$. Свако померање индекса l и r постаје сложености $O(n)$, па пошто се индекси померају $O(n)$ пута, укупна сложеност алгоритама постаје $O(n \log n)$, што је прилично ефикасно у односу на сва ранија решења.

```

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

/* Fenvikovo drvo */

// uvecava se element na poziciji k Fenvikovog drveta za vrednost v i
// azurira se drvo
void uvecaj(vector<int>& drvo, int k, int v) {
    while (k < drvo.size()) {
        drvo[k] += v;
        k += k & -k;
    }
}

```

```

}

// ubacuje element k u Fenvikovo drvo koje predstavlja skup
void ubaci(vector<int>& drvo, int k) {
    uvecaj(drvo, k, +1);
}

// izbacuje element k iz Fenvikovog drveta koje predstavlja skup
void izbacij(vector<int>& drvo, int k) {
    uvecaj(drvo, k, -1);
}

// zbir prefiksa [1, k] Fenvikovog drveta
long long zbirPrefiksa(const vector<int>& drvo, int k) {
    long long zbir = 0;
    while (k > 0) {
        zbir += drvo[k];
        k -= k & -k;
    }
    return zbir;
}

// zbir sufiksa (k, n] Fenvikovog drveta
long long zbirSufiksa(const vector<int>& drvo, int k) {
    return zbirPrefiksa(drvo, drvo.size() - 1) - zbirPrefiksa(drvo, k);
}

// broj elemenata u skupu predstavljenim Fenvikovim drvetom koji su
// strogo veci od vrednosti k
long long brojVecihOd(int k, const vector<int>& drvo) {
    return zbirSufiksa(drvo, k);
}

// broj elemenata u skupu predstavljenim Fenvikovim drvetom koji su
// strogo manji od vrednosti k
long long brojManjihOd(int k, const vector<int>& drvo) {
    return zbirPrefiksa(drvo, k-1);
}

int main() {
    // ucitavamo ulazne podatke
    long long maksInverzija;
    cin >> maksInverzija;
    int n;
    cin >> n;
    vector<int> a(n);
    for(int i = 0; i < n; i++)
        cin >> a[i];

    // odredjujemo rang svakog elementa da bismo elemente mogli da
    // smestamo u Fenvikovo drvo
    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;
}

```

```
// brojimo ukupan broj inverzija polaznog niza
long long brojInverzija = 0;
vector<int> BIT(n+1, 0);
for (int i = 0; i < n; i++) {
    brojInverzija += brojVecihOd(a[i], BIT);
    ubaci(BIT, a[i]);
}

// trazeni broj parova (l, r)
long long brojParova = 0;

// krecemo od para (0, 1)
int l = 0, r = 1;

// skup elemenata na pozicijama [0, L]
vector<int> BITDoL(n+1, 0);
ubaci(BITDoL, a[0]);
// skup elemenata na pozicijama [R, n)
vector<int> BITOdR = BIT;
izbaci(BITOdR, a[0]);

// za svaku mogucu vrednost desnog kraja
while (r < a.size()) {
    // odredjujemo poslednju vrednost l takvu da je broj inverzija kada se
    // izbace elementi iz intervala (l, r) manji od zadate granice
    while (true) {
        // posto mora da vazi l < r, l vise nije moguće uvecati
        if (l+1 == r)
            break;
        // ispitujemo da li je moguće povećati l, tj. ubaciti element a[l+1]
        long uvecanje = brojVecihOd(a[l+1], BITDoL) +
            brojManjihOd(a[l+1], BITOdR);
        // ako nije, trenutno l je poslednje moguće
        if (brojInverzija + uvecanje > maksInverzija)
            break;
        // u suprotnom ubacujemo a[l+1] u skup, azuriramo broj inverzija
        // i povećavamo l
        brojInverzija += uvecanje;
        ubaci(BITDoL, a[l+1]);
        l++;
    }

    // ako je broj inverzija za trenutni par (l, r) ispod dopustene
    // granice, takav je i broj inverzija za sve parove
    // (0, r), (1, r), ..., (l, r)
    if (brojInverzija <= maksInverzija)
        brojParova += l + 1;

    // uvecavamo r tako sto izbacujemo a[r] iz skupa i azuriramo broj
    // inverzija
    long umanjenje = brojVecihOd(a[r], BITDoL) +
        brojManjihOd(a[r], BITOdR);
    brojInverzija -= umanjenje;
    izbaci(BITOdR, a[r]);
    r++;
}

// ispisujemo konacan rezultat
```

```

cout << brojParova << endl;

return 0;
}

```

Задатак: Број различитих елемената у сегментима

Напиши програм који одређује број различитих елемената у сваком од m датих сегмената низа од n елемената.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 50000$), а затим n целих бројева $a_1, \dots, a_n$, чије су вредности између 1 и 100000. Након тога се учитава број m ($1 \leq m \leq 50000$) а затим у наредних m редова лева и десна граница затвореног сегмента $[l_i, d_i]$, раздвојене са по једним размаком ($1 \leq l_i \leq d_i \leq n$).

Излаз: На стандардни излаз исписати m бројева од којих сваки представља број различитих елемената у сегменту $a_{l_i}, \dots, a_{d_i}$.

Пример

Улаз	Излаз
5	3
1 1 2 1 3	2
3	3
1 5	
2 4	
3 5	

Решење

Решење грубом силом подразумева да за сваки сегмент избројимо различите елементе (на пример, коришћењем структуре података за чување скупова). Оно је прилично неефикасно.

```

#include <iostream>
#include <vector>
#include <unordered_set>

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        int l, d;
        cin >> l >> d;
        unordered_set<int> elementi;
        for (int j = l; j <= d; j++)
            elementi.insert(a[j]);
        cout << elementi.size() << endl;
    }
    return 0;
}

```

Основна идеја ефикасног решења задатка је да ако знамо позиције последњих појављивања елемената низа унутар неког сегмента позиција $[1, d]$ тада можемо лако одредити број различитих елемената у сваком од сегмената $[l, d]$ као број таквих позиција које припадају том сегменту. Можемо креирати низ дужине d који садржи јединице на местима на којима се елемент јавља последњи пут у сегменту и нуле на местима на којима се налазе елементи који се у том сегменту јављају и касније. На пример, ако је дато 10 елемената низа 3, 4, 1, 3, 2, 5, 4, 7, 2, 3, тада можемо направити низ 0, 0, 1, 0, 0, 1, 1, 1, 1, 1. За сваки сегмент облика $[l, d]$

(за фиксиран десни крај d) број различитих елемената можемо добити израчунавањем броја јединица у одговарајућем сегменту низа нула и јединица. Заиста, за сваку групу једнаких елемената само постоји само једна јединица и сваки елемент се броји само једном. За сваки елемент који припада сегменту $[l, d]$ постоји бар једна јединица у том сегменту која му одговара (ако је тренутно појављивање елемента последње, онда је јединица на његовом месту, а ако није, онда сигурно постоји јединица негде иза њега која му одговара). Приметимо да то не мора да важи за сегменте који се не завршавају на позицији d .

Низ нула и јединица можемо одржавати инкрементално. Наиме, за сваки нови елемент полазног низа на крај низа додајемо јединицу (ово његово појављивање је сигурно последње). Додатно, можемо проверити да ли се раније појављивао и ако јесте пронаћи позицију његовог ранијег последњег појављивања и променити јединицу на тој позицији у нулу. Позиције последњих појављивања елемената можемо чувати у засебној мапи.

Ово нам указује на то да би добро било да унете сегменте сортирамо на основу десних крајева и обрађивати их у том редоследу. Низ нула и јединица можемо чувати у Фенвиковом или сегментном стаблу, што нам омогућава да ефикасно ажурирамо појединачне вредности и одређујемо збирове његових суфикса.

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

using namespace std;

// операције за рад са Фенвиковим дрветом

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 zbirSegmenta(const vector<int>& drvo, int l, int d) {
    return zbirPrefiksa(drvo, d) - zbirPrefiksa(drvo, l-1);
}

int main() {
    ios_base::sync_with_stdio(false);
    int n;
    cin >> n;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++)
        cin >> a[i];

    int m;
    cin >> m;

    struct Upit {
        int l, d, i;
```



```

};

vector<Upit> upiti(m);
for (int i = 0; i < m; i++) {
    cin >> upiti[i].l >> upiti[i].d;
    upiti[i].i = i;
}

sort(begin(upiti), end(upiti),
    [](const auto& u1, const auto& u2) {
        return u1.d < u2.d;
    });

unordered_map<int, int> prethodna_pozicija;
vector<int> drvo(n + 1, 0);
vector<int> rezultat(m);
int tekuci_upit = 0;
for (int i = 1; tekuci_upit < upiti.size() && i <= n; i++) {
    auto it = prethodna_pozicija.find(a[i]);
    if (it != prethodna_pozicija.end()) {
        dodaj(drvo, it->second, -1);
        it->second = i;
    } else {
        prethodna_pozicija[a[i]] = i;
    }
    dodaj(drvo, i, 1);
    while (tekuci_upit < upiti.size() && upiti[tekuci_upit].d == i) {
        rezultat[upiti[tekuci_upit].i] =
            zbirSegmenta(drvo, upiti[tekuci_upit].l, i);
        tekuci_upit++;
    }
}

for (int x : rezultat)
    cout << x << '\n';

return 0;
}

```

5.3.4 Лења сегментна дрвета

Видели смо да и сегментна и Фенвикова дрвета подржавају ефикасно израчунавање статистика одређених сегмената (распона) низа и ажурирање појединачних елемената низа. Ажурирање целих сегмената низа одједном није директно подржано. Ако се оно сведе на појединачно ажурирање свих елемената унутар сегмента, добија се лоша сложеност.

Могуће је једноставно употребити Фенвиково дрво тако да се ефикасно подржи увећавање свих елемената из датог сегмента одједном, али онда се губи могућност ефикасног израчунавања збирова елемената сегмената, већ је само могуће ефикасно враћати вредности појединачних елемената полазног низа. Основна идеја је да се одржава низ разлика суседних елемената полазног низа и да се тај низ разлика чува у Фенвиковом дрвету. Увећавање свих елемената сегмената полазног низа за неку вредност x , своди се на промену два елемента низа разлика, док се реконструкција елемента полазног низа на основу низа разлика своди на израчунавање збира одговарајућег префикса, што се помоћу Фенвиковог дрвета може урадити веома ефикасно. На овај начин и ажурирање целих сегмената низа одједном и читавање појединачних елемената можемо постићи у сложености $O(\log n)$, што није било могуће само уз коришћење низа разлика (увећавања свих елемената неког сегмента је тада било сложености $O(1)$, али је читавање вредности из низа било сложености $O(n)$).

Ефикасно увећање свих елемената у датом сегменту за исту вредности и израчунавање збирова сегмената могуће је имплементирати помоћу одржавања два Фенвикова дрвета или помоћу тзв. *сејменјиних дрвећа са лењом пропацијом* (енгл. *lazy propagation*). Ове технике ћемо описати у наредном задатку.

Задатак: Увећања и зборови сегмената

Напиши програм који читава низ елемената, а затим да се више пута сви елементи низа у неком распону позиција увећају за одређену вредност и да се израчунају вредности задатих сегмената низа.

Улаз: У првој линији стандардног улаза налази се број n ($1 \leq n \leq 100000$), а у наредној линији низ од n елемената (елементи су цели бројеви између 0 и 10, раздвојени са по једним размаком). У наредној линији налази се број m ($1 \leq m \leq 100000$), а у наредних m линија упити. Подржане су две врсте упита:

- $u\ a\ b\ v$ – извршавање овог упита подразумева да се сви елементи низа на позицијама $[a, b]$ увећају за вредност v ($0 \leq i < n$, $-10 \leq v \leq 10$).
- $z\ a\ b$ – извршавање овог упита подразумева да се израчуна и на стандардни излаз испише збир елемената низа који су на позицијама $[a, b]$.

Излаз: Стандардни излаз садржи резултате z упита (сваки у посебној линији).

Пример

Улаз	Излаз
5	15
1 2 3 4 5	20
7	26
z 0 4	20
u 0 4 1	13
z 0 4	
u 1 3 2	
z 0 4	
z 0 3	
z 3 4	

Решење

Решење грубом силом подразумева да се и увећавање и израчунавање збирова сегмената врши у сложености $O(n)$, па је укупна сложеност $O(mn)$.

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

using namespace std;

int main() {
    int n; cin >> n;
    vector<int> niz(n);
    for (int i = 0; i < n; i++)
        cin >> niz[i];
    int m; cin >> m;
    for (int i = 0; i < m; i++) {
        string upit;
        cin >> upit;
        if (upit == "u") {
            int a, b, v;
            cin >> a >> b >> v >> ws;
            for (int j = a; j <= b; j++)
                niz[j] += v;
        } else if (upit == "z") {
            int a, b;
            cin >> a >> b;
            int zbir = 0;
            for (int j = a; j <= b; j++)
                zbir += niz[j];
            cout << zbir << '\n';
        }
    }
}
```

```
    return 0;
}
```

Ефикасније решење можемо добити помоћу сегментног дрвета са лењом пропагацијом.

За лењу пропагацију нам је битно да знамо рад са сегментним дрветом одозго наниже.

Сваки чвор у сегментном дрвету се односи на одређени сегмент елемената полазног низа и чува збир тог сегмента. Ако се тај сегмент у целости садржи унутар сегмента који се ажурира, можемо унапред израчунати за колико се повећава вредност у корену. Наиме, пошто се свака вредност у сегменту повећава за v , тада се вредност збира тог сегмента повећава за $k \cdot v$, где је k број елемената у том сегменту. Вредност збира у корену тиме бива ажурирана у константном времену, али вредности збирова унутар поддрвета којима је то корен (укључујући и вредности у листовима које одговарају вредностима полазног низа) и даље остају неажурне. Њихово ажурирање захтевало би линеарно време, што нам је недопустиво. Кључна идеја је да се ажурирање тих вредности одложи и да се оне не ажурирају одмах, већ само током неке касније посете тим чворовима, до које би дошло и иначе (не желимо да те чворове посећујемо само због овог ажурирања, већ ћемо ажурирање урадити успут, током неке друге посете тим чворовима која би се свакако морала десити). Поставља се питање како да сигнализирамо да вредности збирова у неком поддрвету нису ажурне и додатно оставимо упутство на који начин се могу ажурирати. У том циљу проширујемо чворове и у сваком од њих поред вредности збира сегмента чувамо и додатни *коэффициент лење пропагације*. Ако дрво у свом корену има коефицијент лење пропагације s који је различит од нуле, то значи да вредности збирова у целом том дрвету нису ажурне и да је сваки од листова тог дрвета потребно повећати за s и у односу на то ажурирати и вредности збирова у свим унутрашњим чворовима тог дрвета (укључујући и корен). Ажурирање се може одлагати све док вредност збира у неком чвору не постане заиста неопходна, а то је тек приликом упита израчунавања вредности збира неког сегмента. Ипак, вредности збирова у чворовима ћемо ажурирати и чешће и то заправо приликом сваке посете чвору – било у склопу операције увећања вредности из неког сегмента позиција полазног низа, било у склопу упита израчунавања збира неког сегмента. На почетку обе рекурзивне функције ћемо проверавати да ли је вредност коефицијента лење пропагације различита од нуле и ако јесте, ажурираћемо вредност збира тако што ћемо га увећати за производ тог коефицијента и броја елемената у сегменту који тај чвор представља, а затим коефицијент лење пропагације оба његова детета увећати за тај коефицијент (тиме корен дрвета који тренутно посећујемо постаје ажуран, а његовим поддрветима се даје упутство како их у будућности ажурирати). Приметимо да се избегава ажурирање целог дрвета одједном, већ се ажурира само корен, што је операција сложености $O(1)$.

Имајући ово у виду, размотримо како се може имплементирати функција која врши увећање свих елемената неког сегмента. Њена инваријанта ће бити да сви чворови у дрвету или садрже ажурне вредности збирова или ће бити исправно обележени за каснија ажурирања (преко коефицијента лење пропагације), а да ће након њеног извршавања корен дрвета на ком је позвана садржати актуелну вредност збира. Након почетног обезбеђивања да вредност у текућем чвору постане ажурна, могућа су три случаја. - Ако је сегмент у текућем чвору дисјунктан у односу на сегмент који се ажурира, тада су сви чворови у поддрвету којем је он корен већ или ажурни или исправно обележени за касније ажурирање и није потребно ништа урадити. - Ако је сегмент који одговара текућем чвору потпуно садржан у сегменту чији се елементи увећавају, тада се његова вредност ажурира (увећавањем за $k \cdot v$, где је k број елемената сегмента који одговара текућем чвору, а v вредност увећања), а његовој деци се коефицијент лење пропагације увећава за v . - Ако се два сегмента секу, тада се прелази на рекурзивну обраду два детета. Након извршавања функције над њима, сигурни смо да ће сви чворови у левом и десном поддрвету задовољавати услов инваријанте и да ће оба корена имати ажурне вредности. Ажурну вредност у корену постигаћемо сабирањем вредности два детета.

Прикажимо рад ове функције на једном примеру.

				26/0			
		10/0			16/0		
	7/0		3/0		11/0		5/0
3/0	4/0	1/0	2/0	6/0	5/0	1/0	4/0
0	1	2	3	4	5	6	7

Прикажимо како бисмо све елементе из сегмента позиција $[2, 7]$ увећали за 3. Сегменти $[0, 7]$ и $[2, 7]$ се секу, па стога ажурирање препуштамо наследницима и након њиховог ажурирања, при повратку из рекурзије вредност одређује као збир његових ажурираних вредности. На левој страни се сегмент $[0, 3]$ сече са $[2, 7]$ па и он препушта ажурирање наследницима и ажурира се тек при повратку из рекурзије. Сегмент $[0, 1]$ је дисјунктан у односу на $[2, 7]$ и ту онда није потребно ништа радити. Сегмент $[2, 3]$ је цео садржан у $[2, 7]$, и код њега

директно можемо да знамо како се збир увећава. Пошто имамо два елемента и сваки се увећава за 3, збир се увећава укупно за $2 \cdot 3 = 6$ и поставља се на 9. У овом тренутку избегавамо ажурирање свих вредности у дрвету у ком је тај елемент корен, већ само наследницима уписујемо да је потребно пропагирати увећање за 3, али саму пропагацију одлажемо за тренутак када она постане неопходна. У повратку из рекурзије, вредност 10 увећавамо на $7 + 9 = 16$. Што се тиче десног подрвета, сегмент $[4, 7]$ је цео садржан у сегменту $[2, 7]$, па и ту можемо израчунати вредност збира. Пошто се 4 елемента увећавају за по 3, укупан збир се увећава за $4 \cdot 3 = 12$. Зато се вредност 16 мења у 28. Пропагацију ажурирања кроз подрво одлажемо и само његовој деци бележимо да је увећање за 3 потребно извршити у неком каснијем тренутку. При повратку из рекурзије вредност у корену ажурирамо са 26 на $16 + 28 = 44$. Након тога добија се следеће дрво.

				44/0			
		16/0				28/0	
7/0		9/0		11/3		5/3	
3/0	4/0	1/3	2/3	6/0	5/0	1/0	4/0
0	1	2	3	4	5	6	7

Претпоставимо да се сада елементи из сегмента $[0, 5]$ увећавају за 2. Поново се креће од врха и када се установи да се сегмент $[0, 7]$ сече са $[0, 5]$ ажурирање се препушта деци и вредност се ажурира тек при повратку из рекурзије. Сегмент $[0, 3]$ је цео садржан у $[0, 5]$, па се зато вредност 16 увећава за $4 \cdot 2 = 8$ и поставља на 24. Подрвета се не ажурирају одмах, већ се само њиховим коренима уписује да је све вредности потребно ажурирати за 2. У десном подрвету сегмент $[4, 7]$ се сече са $[0, 5]$, па се рекурзивно обрађују подрвета. При обради чвора 11, примећује се да је он требао да буде ажуриран, међутим, још није, па се онда његова вредност ажурира и увећава за $2 \cdot 3$ и са 11 мења на 17. Његови наследници се не ажурирају одмах, већ само ако то буде потребно и њима се само уписује леђа вредност 3. Тек након тога се примећује да се сегмент $[4, 5]$ цео садржи у $[0, 5]$, па се онда вредност 17 увећава за $2 \cdot 2 = 4$ и поставља на 21. Подрвета се не ажурирају одмах, већ само по потреби тако што се у њиховим коренима постави вредност лењог коефицијена. Пошто је у њима већ уписана вредност 5 и пошто оно није ажурно, прво се та вредност 5 увећава за $2 \cdot 3 = 6$ и поставља на 11, а његовој деци се лењи коефицијент поставља на 3. Након тога се примећује да је сегмент $[6, 7]$ дисјунктан са $[0, 5]$ и не ради се ништа. У повратку кроз рекурзију се ажурирају вредности родитељских чворова и долази се до наредног дрвета.

				56/0			
		24/0				32/0	
7/2		9/2		21/0		11/0	
3/0	4/0	1/3	2/3	6/5	5/5	1/3	4/3
0	1	2	3	4	5	6	7

Функција израчунавања вредности збира сегмента остаје практично непромењена, осим што се при уласку у сваки чвор врши његово ажурирање. Прикажимо сада такве функције на текућем примеру. Размотримо како се сада израчунава збир елемената у сегменту $[3, 5]$. Крећемо од врха. Сегмент $[0, 7]$ се сече са $[3, 5]$, па се рекурзивно обрађују деца. У левом подрвету сегмент $[0, 3]$ такође има пресек са $[3, 5]$ па прелазимо на наредни ниво рекурзије. Приликом посете чвора у чијем је корену вредност 7 примећује се да његова вредност није ажурна, па се користи прилика да се она ажурира, тако што се увећа за $2 \cdot 2 = 4$, а наследницима се лењи коефицијент увећа за 2. Пошто је сегмент $[0, 1]$ дисјунктан са $[3, 5]$, враћа се вредност 0. Приликом посете чвора у чијем је корену вредност 9 примећује се да његова вредност није ажурна, па се користи прилика да се она ажурира, тако што се увећа за $2 \cdot 2 = 4$, а наследницима се лењи коефицијент увећа за 2. Сегмент $[2, 3]$ се сече са $[3, 5]$, па се рекурзивно врши обрада подрвета. Вредност 1 се прво ажурира тако што се повећа за $1 \cdot 5 = 5$, а онда, пошто је $[2, 2]$ дисјунктно са $[3, 5]$ враћа се вредност 0. Вредност 2 се такође прво ажурира тако што се повећа за $1 \cdot 5 = 5$, а пошто је сегмент $[3, 3]$ потпуно садржан у $[3, 5]$ враћа се вредност 7. У десном подрвету је чвор са вредношћу 32 ажуран, сегмент $[4, 7]$ се сече са $[3, 5]$, па се прелази на обраду наследника. Чвор са вредношћу 21 је ажуран, сегмент $[4, 5]$ је цео садржан у $[3, 5]$, па се враћа вредност 21. Чвор са вредношћу 11 је такође ажуран, али је сегмент $[6, 7]$ дисјунктан у односу на $[3, 5]$, па се враћа вредност 0. Дакле, чворови 13 и 24 враћају вредност 7, чвор 32 враћа вредност 21, па чвор 56 враћа вредност 28. Стање дрвета након извршавања упита је следеће.

				56/0			
		24/0				32/0	
11/0		13/0		21/0		11/0	
3/2	4/2	6/0	7/0	6/5	5/5	1/3	4/3

```

0      1      2      3      4      5      6      7

#include <iostream>
#include <vector>

using namespace std;

// ažurira elemente lenjog segmentnog drveta koje je smešteno u nizove
// drvo i lenjo od pozicije k u kome se čuvaju zbirovi elemenata
// originalnog niza sa pozicija iz segmenta [x, y] nakon što su u
// originalnom nizu svi elementi sa pozicija iz segmenta [a, b]
// uvećani za vrednost v
void uvecaj(vector<int>& drvo, vector<int>& lenjo, int k, int x, int y,
            int a, int b, int v) {
    // ažuriramo vrednost u korenu, ako nije ažurna
    if (lenjo[k] != 0) {
        drvo[k] += (y - x + 1) * lenjo[k];
        // ako nije u pitanju list propagaciju prenosimo na decu
        if (x != y) {
            lenjo[2*k] += lenjo[k];
            lenjo[2*k+1] += lenjo[k];
        }
        lenjo[k] = 0;
    }
    // ako su intervali disjunktni, ništa nije potrebno raditi
    if (b < x || y < a) return;
    // ako je interval [x, y] ceo sadržan u intervalu [a, b]
    if (a <= x && y <= b) {
        drvo[k] += (y - x + 1) * v;
        // ako nije u pitanju list propagaciju prenosimo na decu
        if (x != y) {
            lenjo[2*k] += v;
            lenjo[2*k+1] += v;
        }
    } else {
        // u suprotnom se intervali seku,
        // pa rekurzivno obilazimo poddrveta
        int s = (x + y) / 2;
        uvecaj(drvo, lenjo, 2*k, x, s, a, b, v);
        uvecaj(drvo, lenjo, 2*k+1, s+1, y, a, b, v);
        drvo[k] = drvo[2*k] + drvo[2*k+1];
    }
}

// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [0, n-1]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void uvecaj(vector<int>& drvo, vector<int>& lenjo,
            int a, int b, int v) {
    int n = drvo.size() / 2;
    uvecaj(drvo, lenjo, 1, 0, n-1, a, b, v);
}

// na osnovu lenjog segmentnog drveta koje je smešteno u nizove drvo i
// lenjo od pozicije k u kome se čuvaju zbirovi elemenata polaznog
// niza sa pozicija iz segmenta [x, y] izračunava se zbir elemenata

```

```

// polaznog niza sa pozicija iz segmenta [a, b]
int zbirSegmenta(vector<int>& drvo, vector<int>& lenjo,
                int k, int x, int y, int a, int b) {
    // ažuriramo vrednost u korenu, ako nije ažurna
    if (lenjo[k] != 0) {
        drvo[k] += (y - x + 1) * lenjo[k];
        if (x != y) {
            lenjo[2*k] += lenjo[k];
            lenjo[2*k+1] += lenjo[k];
        }
        lenjo[k] = 0;
    }

    // intervali [x, y] i [a, b] su disjunktni
    if (b < x || a > y) return 0;
    // interval [x, y] je potpuno sadržan unutar intervala [a, b]
    if (a <= x && y <= b)
        return drvo[k];
    // intervali [x, y] i [a, b] se seku
    int s = (x + y) / 2;
    return zbirSegmenta(drvo, lenjo, 2*k, x, s, a, b) +
        zbirSegmenta(drvo, lenjo, 2*k+1, s+1, y, a, b);
}

// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [0, n-1]
// izračunava se zbir elemenata polaznog niza sa pozicija
// iz segmenta [a, b]
int zbirSegmenta(vector<int>& drvo, vector<int>& lenjo, int a, int b) {
    // računamo doprinos celog niza,
    // tj. elemenata iz intervala [0, n-1]
    int n = drvo.size() / 2;
    return zbirSegmenta(drvo, lenjo, 1, 0, n-1, a, b);
}

// od elemenata niza a sa pozicija [x, y]
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije k
void formirajDrvo(const vector<int>& a, vector<int>& drvo,
                 size_t k, size_t x, size_t y) {
    if (x == y)
        // u listove prepisujemo elemente polaznog niza
        drvo[k] = x < a.size() ? a[x] : 0;
    else {
        // rekursivno formiramo levo i desno poddrvo
        int s = (x + y) / 2;
        formirajDrvo(a, drvo, 2*k, x, s);
        formirajDrvo(a, drvo, 2*k+1, s+1, y);
        // izračunavamo vrednost u korenu
        drvo[k] = drvo[2*k] + drvo[2*k+1];
    }
}

// najmanji stepen dvojke veci ili jednak od n
int stepenDvojke(int n) {
    int s = 1;
    while (s < n)

```

```

    s <= 1;
    return s;
}

// na osnovu datog niza a duzine n u kom su elementi smesteni od
// pozicije 0 formira se segmentno drvo i elementi mu se smeštaju u
// niz drvo krenuvši od pozicije 1
void formirajDrvo(const vector<int>& a, vector<int>& drvo, vector<int>& lenjo) {
    // niz implicitno dopunjujemo nulama tako da mu duzina postane
    // najblizi stepen dvojke
    int n = stepenDvojke(a.size());
    drvo.resize(n * 2);
    lenjo.resize(n * 2, 0);
    // krećemo formiranje od korena koji se nalazi u nizu drvo
    // na poziciji 1 i pokriva elemente na pozicijama [0, n-1]
    formirajDrvo(a, drvo, 1, 0, n - 1);
}

int main() {
    int n; cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++)
        cin >> a[i];
    vector<int> drvo, lenjo;
    formirajDrvo(a, drvo, lenjo);
    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        string upit;
        cin >> upit;
        if (upit == "u") {
            int a, b, v;
            cin >> a >> b >> v >> ws;
            uvecaj(drvo, lenjo, a, b, v);
        } else if (upit == "z") {
            int a, b;
            cin >> a >> b;
            cout << zbirSegmenta(drvo, lenjo, a, b) << '\n';
        }
    }

    return 0;
}

```

Задатак је могуће ефикасно решити и применом Фенвикових дрвета. Збир сваког сегмента $[a, b]$ увећаног низа разложићемо на збир сегмента $[a, b]$ полазног низа и збир свих увећања елемената сегмента $[a, b]$.

Пошто се оригинални низ не мења, збирове сегмената оригиналног низа можемо израчунати коришћењем збирова префикса (као у задатку **Збирови сегмената**).

Да бисмо могли ефикасно да израчунавамо збирове увећања користићемо следећу идеју. Замислимо да одржавамо низ увећања свих елемената. Он је иницијализован нулама и приликом сваког увећања неког сегмента оригиналног низа увећава се и његов одговарајући сегмент. На пример, након увећања сегмента $[a, b]$ у низ увећања има следећи облик.

U_0	$\dots$	U_{a-1}	U_a	U_{a+1}	$\dots$	U_b	U_{b+1}	$\dots$	U_{n-1}
0	$\dots$	0	v	v	$\dots$	v	0	$\dots$	0

Размотримо шта се дешава са префиксним збировима увећања. Познавање ових префиксних збирова нам омогућавају да лако израчунамо збир сваког сегмента $[l, d]$ и то као $Z_{d+1} - Z_l$. Нека је $Z_0 = 0$ и $Z_k = \sum_{i=0}^{k-1} U_i = U_0 + \dots + U_{k-1}$.

Z_0	$\dots$	Z_a	Z_{a+1}	$\dots$	Z_b	Z_{b+1}	Z_{b+2}	$\dots$	Z_n
0	$\dots$	0	v	$\dots$	$(b-a)v$	$(b-a+1)v$	$(b-a+1)v$	$\dots$	$(b-a+1)v$

Зборови префикса Z_k увећања се могу разложити на два дела, тј. за свако $0 \leq k \leq n$ можемо дефинисати X_k и Y_k тако да је $Z_k = X_k + Y_k$.

X_0	$\dots$	X_a	X_{a+1}	$\dots$	X_b	X_{b+1}	X_{b+2}	$\dots$	X_n
0	$\dots$	0	$(a+1)v$	$\dots$	bv	$(b+1)v$	0	$\dots$	0
Y_0	$\dots$	Y_a	Y_{a+1}	$\dots$	Y_b	Y_{b+1}	Y_{b+2}	$\dots$	Y_n
0	$\dots$	0	$-av$	$\dots$	$-av$	$-av$	$(b-a+1)v$	$\dots$	$(b-a+1)v$

Можемо приметити да је $0 \leq k \leq n$ важи да је $X_k = k\bar{U}_k$, где је $\bar{U}_0 = 0$ и за $1 \leq k \leq n$ је $\bar{U}_k = U_{k-1}$. Вредности тога низа су следеће.

$\bar{U}_0$	$\dots$	$\bar{U}_a$	$\bar{U}_{a+1}$	$\dots$	$\bar{U}_b$	$\bar{U}_{b+1}$	$\bar{U}_{b+2}$	$\dots$	$\bar{U}_n$
0	$\dots$	0	v	$\dots$	v	v	0	$\dots$	0

Стога је за израчунавање вредности Z_k довољно познавати вредности низова $\bar{U}$ и Y . Директно ажурирање ова два низа приликом увећања сегмента захтева линеарно време. Међутим, пошто су суседни елементи ових низова у највећој мери једнаки, ефикасније је одржавати их одржавањем низова разлика њихових суседних елемената (елементи ових низова се лако могу реконструисати израчунавањем префиксних сума следећих низова разлика). Обележимо са $u_k = \bar{U}_{k+1} - \bar{U}_k$, за $0 \leq k < n$ низ разлика низа $\bar{U}$. Слично, обележимо са $y_k = Y_{k+1} - Y_k$, за $0 \leq k < n$ низ разлика низа Y .

u_0	$\dots$	u_{a-1}	u_a	u_{a+1}	$\dots$	u_b	u_{b+1}	$\dots$	u_{n-1}
0	$\dots$	0	v	0	$\dots$	0	$-v$	$\dots$	0
y_0	$\dots$	y_{a-1}	y_a	y_{a+1}	$\dots$	y_b	y_{b+1}	$\dots$	y_{n-1}
0	$\dots$	0	$-av$	0	$\dots$	0	$(b+1)v$	$\dots$	0

За $1 \leq k \leq n$ важи да је $\bar{U}_k = \sum_{i=0}^{k-1} u_i = u_0 + \dots + u_{k-1}$ и да је $Y_k = \sum_{i=0}^{k-1} y_i = y_0 + \dots + y_{k-1}$. Да бисмо могли да ефикасно израчунавамо њихове префиксне збирове, оба низа можемо одржавати у Фенвиковим дрветима. Тада познајући вредности низова u и y веома једноставно можемо одредити било који парцијални збир Z_k . У времену $O(\log n)$ можемо одредити $\bar{U}_k$ и Y_k , и затим $X_k = k\bar{U}_k$ и $Z_k = X_k + Y_k$.

Дакле, приликом увећања сваког елемента сегмента $[a, b]$ оригиналног низа за вредност v , у првом Фенвиковом дрвету у ком чувамо низ u елемент на позицији a увећавамо за v , елемент на позицији $b+1$ за вредност $-v$, док у другом дрвету у ком чувамо низ y елемент на позицији a увећавамо за вредност $-av$, а елемент на позицији $b+1$ за вредност $(b+1) \cdot v$. Имајући у виду особине Фенвиковог дрвета, све ово се остварује у времену $O(\log n)$.

Приликом израчунавања укупног збира увећања на позицијама сегмента $[l, d]$ рачунамо $Z_{d+1} - Z_l$. Да бисмо израчунали Z_{d+1} израчунавамо $(d+1) \cdot \bar{U}_{d+1} - Y_{d+1}$, а да бисмо израчунали Z_l израчунавамо $l \cdot \bar{U}_l - Y_l$, при чему се $\bar{U}_{d+1}$ израчунава као збир првих d елемената низа u , $\bar{U}_l$ израчунава као збир првих $l-1$ елемената низа u , Y_{d+1} израчунава као збир првих d елемената низа y , а Y_l израчунава као збир првих $l-1$ елемената низа y .

Докажимо да се ова веза између низова U , u , y и Z одржава након сваког увећања низа. Претпоставимо да су низови $\bar{U}$, X , Y и Z дефинисани као у претходној дискусији. Претпостављамо, дакле, да важи да је за $1 \leq k \leq n$ $k\bar{U}_k + Y_k = \sum_{i=0}^{k-1} U_i$.

Претпоставимо да се у низу увећава сегмент $[a, b]$. Тада за свако $a \leq k \leq b$ важи $U'_k = U_k + v$, док је остале

вредности k важи $U'_k = U_k$. Такође, $u'_a = u_a + v$, $u'_{b+1} = u_{b+1} - v$, да је $y'_a = y_a - av$, $y'_{b+1} = y_{b+1} + (b+1)v$, док за све остале индексе k важи да је $u'_k = u_k$ и $y'_k = y_k$. Докажимо да за свако $1 \leq k \leq n$ важи $k\bar{U}'_k + Y'_k = \sum_{i=0}^{k-1} U'_i$. Размотримо следеће случајеве (случај $k = 0$ је тривијалан јер у њему сви низови имају вредност 0).

- Ако је $1 \leq k \leq a$, тада је $k\bar{U}'_k + Y'_k = k(\sum_{i=0}^{k-1} u'_i) + \sum_{i=0}^{k-1} y'_i = k(\sum_{i=0}^{k-1} u_i) + \sum_{i=0}^{k-1} y_i = k\bar{U}_k + Y_k = \sum_{i=0}^{k-1} U_i = \sum_{i=0}^{k-1} U'_i$.
- Ако је $a < k \leq b$ тада је $k\bar{U}'_k + Y'_k = k(\sum_{i=0}^{k-1} u'_i) + \sum_{i=0}^{k-1} y'_i = k(\sum_{i=0}^{k-1} u_i) + kv + \sum_{i=0}^{k-1} y_i - av = \sum_{i=0}^{k-1} U_i + (k-a)v = \sum_{i=0}^{a-1} U_i + \sum_{i=a}^{k-1} (U_i + v) = \sum_{i=0}^{k-1} U'_i$.
- На крају, ако је $b < k \leq n$, $k\bar{U}'_k + Y'_k = k(\sum_{i=0}^{k-1} u'_i) + \sum_{i=0}^{k-1} y'_i = k(\sum_{i=0}^{k-1} u_i) + k(v-v) + \sum_{i=0}^{k-1} y_i + (-a+b-1)v = \sum_{i=0}^{k-1} U_i + (b+1-a)v = \sum_{i=0}^{a-1} U_i + \sum_{i=a}^b (U_i + v) + \sum_{i=b+1}^{k-1} U_i = \sum_{i=0}^{k-1} U'_i$.

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

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

```
using namespace std;
```

```
// operacije za rad sa Fenvikovim drvetom
```

```
// azurira Fenvikovo drvo smesteno u niz drvo nakon sto se
// u originalnom nizu element na poziciji k uveca za x
```

```
void dodaj(vector<int>& drvo, int k, int x) {
    k++;
    int n = drvo.size();
    while (k < n) {
        drvo[k] += x;
        k += k & -k;
    }
}
```

```
// na osnovu Fenvikovog drveta smestenog u niz drvo
// izracunava zbir prefiksa [0, k) originalnog niza
```

```
int zbirPrefiksa(const vector<int>& drvo, int k) {
    int zbir = 0;
    while (k > 0) {
        zbir += drvo[k];
        k -= k & -k;
    }
    return zbir;
}
```

```
// azurira dva Fenvikova drveta na osnovu uvecanja elemenata segmenta
// originalnog niza na pozicijama [a, b] za vrednost v
```

```
void uvecaj(vector<int>& drvo1, vector<int>& drvo2,
            int a, int b, int v) {
    dodaj(drvo1, a, v);
    dodaj(drvo1, b+1, -v);
    dodaj(drvo2, a, -a*v);
    dodaj(drvo2, b+1, (b+1)*v);
}
```

```
// na osnovu dva Fenvikova drveta izracunava koliko se uvecao zbir
// elemenata na pozicijama prefiksa [0, k)
```

```
int zbirUvecanjaPrefiksa(const vector<int>& drvo1, const vector<int>& drvo2,
                        int k) {
    return k * zbirPrefiksa(drvo1, k) + zbirPrefiksa(drvo2, k);
}
```

```
}

// na osnovu dva Fenwickova drveta izracunava koliko se uvecao zbir
// elemenata na pozicijama segmenta [a, b]
int zbirUvecanjaSegmenta(const vector<int>& drvo1, const vector<int>& drvo2,
                        int a, int b) {
    return zbirUvecanjaPrefiksa(drvo1, drvo2, b+1) -
           zbirUvecanjaPrefiksa(drvo1, drvo2, a);
}

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0);
    // umesto niza cuvamo zbirove svih njegovih prefiksa
    // sto ce nam omoguciti da brzo odredimo zbirove segmenata
    // originalnog niza
    int n; cin >> n;
    vector<int> zbir(n+1);
    zbir[0] = 0;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        zbir[i+1] = zbir[i] + x;
    }

    // zbirove uvecanja segmenata mozemo brzo odrediti uz pomoc dva
    // segmentna drveta
    vector<int> drvo1(n+1, 0), drvo2(n+2, 0);

    int m;
    cin >> m;
    for (int i = 0; i < m; i++) {
        string upit;
        cin >> upit;
        if (upit == "u") {
            int l, d, v;
            cin >> l >> d >> v >> ws;
            uvecaj(drvo1, drvo2, l, d, v);
        } else if (upit == "z") {
            int l, d;
            cin >> l >> d;
            // zbir segmenata [l, d] uvecanog niza racunamo iz dva dela
            // zbir segmenta polaznog niza i
            int polazniZbir = zbir[d+1] - zbir[l];
            // zbir svih uvecanja tog segmenta
            int zbirUvecanja = zbirUvecanjaSegmenta(drvo1, drvo2, l, d);
            // zbir segmenta [a, b] uvecanog niza
            int zbirSegmenta = polazniZbir + zbirUvecanja;
            cout << zbirSegmenta << '\n';
        }
    }
}
```

Збирка обухвата одабране алгоритме за рад са графовима, анализу текста, алгебарске алгоритме (пре свега из области теорије бројева) и геометријске алгоритме и одабране напредније структуре података (префиксно дрво, структура за представљање унија, Фенвикова и сегментна дрвета).

+

+

+

+

+

+

+

+

+

+

+

+

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 такмичења у средњој школи, у редовној настави у трећем разреду специјализованих ИТ одељења и електротехничких школа, као и у курсевима алгоритама и структура података на факултетима.

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+

+