Dinamičko programiranje

Primer 1: Za dati niz naći njegov najduži neopadajući podniz.

Defnicija: podniz nekog niza je niz koji se dobija izbacivanjem nekih (moguće nijednog) elemenata polaznog niza. Formalno, za niz Z={z[1],z[2],...,z[k]} kažemo da je podniz niza X={x[1],x[2],...,x[n]}, ako postoji strogo rastući niz celih brojeva (indeksa) $i_1,i_2,...,i_k$ takvih da za svako j od 1 do k važi $x[i_j] = z[j]$.

Rešenje:

Kao i ranije, formiranje svih podnizova datog niza (sa pamćenjem najdužeg nerastućeg) je loša ideja jer niz dužine n ima 2^n podnizova, pa bi to bio eksponencijalni algoritam.

Da bismo dobili ideju kako da odredimo elemente niza koji ostaju u podnizu, analizirajmo problem i strukturu njegovog rešenja.

Ako poslednji element x[n] niza X ne učestvuje u najdužem neopadajućem podnizu nnp(X), onda je $nnp(X) = nnp(X_{n-1})$. Ovde X_k (sa velikim X) označava k-ti prefiks niza X, odnosno niz koji se sastoji od prvih k elemenata niza X.

Pretpostavimo sada da x[n] učestvuje kao poslednji element u nnp(X). Neka je pretposlednji član rešenja nnp(X) element x[j]. Želeli bismo da ono što ostane nakon odbacivanja x[n] iz nnp(X), mora da bude nnp niza X_j ili nekog drugog prefiksa niza X, da bismo mogli da zaključimo da problem ima optimalnu podstrukturu. To na žalost to nije tačno, u šta se možemo uveriti na primeru niza X={1, 4, 5, 2, 3, 4}. Očigledno nnp(X) = {1, 2, 3, 4}. Za $nnp(X_5)$ = nnp{1, 4, 5, 2, 3} postoje dve mogućnosti: {1, 4, 5} i {1, 2, 3}. Čak i da na neki način izaberemo onu koja nama odgovara (a to je {1, 2, 3}), odbacivanjem broja 3 dobijamo {1, 2}, što nije nnp ni jednog prefiksa niza X.

Ovde se pojavljuje drugi, mada sličan problem, koji za razliku od polaznog ima optimalnu podstrukturu. Taj problem je da se za dati niz X nađe ***najduži neopadajući podniz koji sadrži poslednji element niza*** – takav podniz ćemo označavati nnpk(X). Zaista, nije teško zaključiti (svođenjem na kontradikciju) da odbacivanjem poslednjeg elementa rešenja ovog problema dobijamo rešenje za neki prefiks niza X. Da bismo od ovog zapažanja imali koristi, treba odgovoriti na dva pitanja:

- Kakva je veza između ova dva problema, odnosno kako nam umeće nalaženja nnpk pomaže da nađemo nnp? Očigledno je da se nnp(X) mora završiti nekim elementom niza X, pa je nnp(X) jednak nekom, tačnije najdužem od svih $nnpk(X_k)$, k=1,n.
- Kako da iskoristimo to što problem nalaženja nnpk ima optimalnu podstrukturu, odnosno kako da nađemo $nnpk(X) = nnpk(X_n)$ kada znamo sve $nnpk(X_k)$ za k<n? Ni ovaj deo problema nije naročito težak: od svih $nnpk(X_k)$ treba izabrati onaj koji završava brojem

manjim ili jednakim x[n] i pri tome je najduži takav; nadovezivanjem x[n] na kraj izabranog nnpk(X_k) dobija se nnpk(X).

Zbog toga što je za nalaženje nnpk(X_n) potrebno znati nnpk(X_k) za sve k=0, n-1, problem ima preklapajuće podprobleme. Ova osobina čini dinamičko programiranje najpogodnijim načinom za rešavanje ovog problema.

U algoritmu koji sledi, d[i] je dužina od nnpk(X_i), a p[i] je indeks pretposlednjeg elementa u nnpk(X_i). Niz p nam je potreban da bismo mogli lako da rekonstruišemo rešenje.

```
OutputNNP(p, x, i)
    if i >= 0
        OutputNNP(p, x, p[i]);
        output(x[i]);

FindNNP(x, n)
    d[0] = 1;
    p[0] = -1;
    imax = 0;
    for i = 1 to n-1
        dmax = 0;
        kmax = -1;
        for k = 0 to n-1
            if (x[k] < x[i]) and (dmax < d[k])
                dmax = d[k];
                kmax = k;
        d[i] = dmax + 1;
        p[i] = kmax;
        if d[imax] < d[i]
            imax = i;
    OutputNNP(p, x, imax);
```

Zanimljivost ovog primera je u tome što je za rešavanje problema *a* bilo potrebno uočiti problem *b*, koji se može rešiti dinamičkim programiranjem, a uz to se rešavanje problema *a* može jednostavno svesti na rešavanje problema *b*.

Primer 2: Dat je niz realnih brojeva *A* od *n* elemenata. Odrediti najmanju od vrednosti svih izraza koji se dobijaju ubacivanjem zagrada u izraz a[1]-a[2]-...-a[n].

Rešenje:

Neka je poslednje oduzimanje koje se izvršava u optimalnom zagrađivanju *k*-to oduzimanje sleva na desno. Onda su u izraz levo od *k*-tog minusa zagrade postavljene tako da on ima minimalnu vrednost, a u izraz desno tako da ima maksimalnu vrednost. Pošto je potrebno da se neki izrazi minimiziraju a neki maksimisiraju, korisno je proširiti početni zadatak, tako da glasi: odrediti raspored zagrada koji minimizira izraz i raspored zagrada koji maksimizira izraz.

Neka je sada *i*-to oduzimanje sleva ono koje se poslednje izvršava pri računanju maksimalne vrednosti izraza. Tada su u prvi deo izraza zagrade tako postavljene da on ima maksimalnu vrednost, a u drugi deo tako da ima minimalnu vrednost. Na osnovu ovog i prethodnog zaključka (o minimumu celog izraza), sledi da proširen problem ima optimalnu podstrukturu. Uvedimo matrice *Min* i *Max*, tako da Min[i][j] i Max[i][j] redom sadrže najmanju, odnosno najveću moguću vrednost koju može imati izraz koji počinje sa a[i], a završava sa a[j]. Ove matrice se mogu izračunavati po dijagonalama, počev od glavne. Preciznije, za j = i+1 važiće

$$\text{Min}[i][j] = \text{Max}[i][j] = a[i]-a[j],$$

a za j > i+1

$$Max[i][j] = \max_{i \le k < j} \{Max[i][k] - Min[k+1][j]\}$$

$$Min[i][j] = \min_{i \le k < j} \{Min[i][k] - Max[k+1][j]\}$$

ili u obliku algoritma:

```
// Popunimo najpre glavnu dijagonalu
for k = 1 to n
    max[k][k] = a[k];
    min[k][k] = a[k];
for q = 1 to n-1
    // Za dijagonalu na rastojanju q od glavne dijagonale, tj. za izraze sa q oduzimanja
    for p = 1 to n-q do
        // za p-ti po redu izraz sa q oduzimanja odredimo minimum i maksimum
        max[p][p+q] = max[p][p] - min[p+1][p+q];
```

```
        min[p][p+q] = min[p][p] - max[p+1][p+q];
    for k = p+1 to p+q-1
            if max[p][k] - min[k+1][p+q] > max[p][p+q]
                max[p][p+q] = max[p,k] - min[k+1][p+q];
            if min[p][k] - max[k+1][p+q] < min[p][p+q]
                min[p][p+q] = min[p][k] - max[k+1][p+q];
```

Primetimo uzgred da, kako se u matricama Min i Max koristi samo deo iznad glavne dijagonale, možemo uštedeti prostor tako što koristimo samo jednu matricu M, tako da je za (i<j) Min[i][j] smešteno u M[i][j], a Max[i][j] u M[j][i].

Tražena minimalna vrednost izraza će se po izvršavanju ovog algoritma nalaziti u Min[1][n]. Ukoliko je potrebno ispisati i sam izraz koji ima minimalnu vrednost, mogu se koristiti dve dodatne matrice (ili jedna uz istu napomenu o uštedi memorije), nazovimo ih imin i imax, koje pamte redni broj oduzimanja koje se poslednje izvršava da bi odgovarajući izraz imao najmanju, odnosno najveću vrednost. Tada bi pomoću jednostavne rekurzivne funkcije iz ovih matrica mogao da se iščita i napiše traženi izraz.

Ovaj primer je zanimljiv iz dva razloga: prvo, videli smo da tabela koju je potrebno popuniti ne mora uvek da se popunjava redom, to jest po vrstama ili po kolonama – ovde je prirodno popunjavati tabelu (matricu) po dijagonalama; i drugo, još zanimljivije, ispostavilo se da je jednostavnije naći maksimume i minimume nego samo minimume. **U nekim problemima se isplati čak proširiti polazni problem i rešiti formalno teži (obuhvatniji) zadatak**, jer se taj novi problem u stvari jednostavnije rešava nego polazni.

U poslednjem primeru se nećemo detaljno baviti analizom, već ćemo samo opisati postupak rešavanja. Primer je interesantan jer matrica rešenja podproblema ne može direktno da se popunjava redom, kako se to obično radi u jednostavnijim zadacima dinamičkog programiranja. Umesto toga, polazni problem se u suštini rastavlja na dva problema koji se rešavaju popunjavanjem tabele redom, a zatim se rešenja problema – delova kombinuju u rešenje polaznog problema.

Primer 3: Data je matrica realnih brojeva *A*, sa *m* redova i *n* kolona. Potrebno odrediti najveći zbir koji se može ostvariti kretanjem kroz matricu. Polazi se sa bilo kog polja u prvom redu, a završetak putanje je bilo koje polje u poslednjem redu. Dozvoljeno je kretati se na dole levo i desno, ali bez ponavljanja polja.

Uvodimo matricu B, u čija polja upisujemo najveći zbir koji se može sakupiti do tog polja, krećući se na dozvoljen način. Radi jednostavnosti algoritma, dodaćemo matrici nulti red i popuniti ga nulama. Na taj način se popunjavanje prvog reda neće razlikovati od popunjavanja ostalih redova.

Ponavljanje polja nije dozvoljeno, pa se u jednom redu matrice ne možemo kretati i nalevo i nadesno, već samo u jednom od ta dva smera. Zato ćemo za svaki red koji nameravamo da popunimo najpre popuniti dva pomoćna niza, L i R, koji će redom sadržati najveće moguće zbirove do odgovarajućih polja u tekućem redu matrice A, krećući se kroz taj red samo na levo (niz L), odnosno samo na desno (niz R). Nakon toga, red matrice B popunjavamo uzimajući za svako njeno polje veći od odgovarajućih zbirova iz nizova L i R.

```
for kol = 1 to n
    B[0][kol] = 0
for red = 1 to m
    L[1] = B[red-1][1] + A[red][1];
    for kol = 2 to n
        L[kol] = max{B[red-1][kol], L[kol-1]} + A[red][kol];
            R[n] = B[red-1][n] + A[red][n];
            for kol = n-1 downto 1
        R[kol] = max{B[red-1][kol], R[kol+1]} + A[red][kol];
    for kol = 1 to n
        B[red][kol] = max{L[kol], R[kol]};
```