Uvod u dinamičko programiranje

Andreja Ilić
e-mail: ilic_andrejko@yahoo.com

Aleksandar Ilić
e-mail: aleksandari@gmail.com

Prirodno Matematički Fakultet u Nišu

1 Uvod

Jedan od čestih algoritamskih problema jeste problem optimizacije: zadati problem može imati više rešenja, svako rešenje ima svoju vrednost, a traži se ono koje ima ekstremnu vrednost. U jednoj širokoj klasi problema optimizacije, rešenje možemo naći korišćenjem **dinamičkog programiranja (eng. dynamic programming)**.

Teorijski gledano probleme optimizacije, kao i probleme pretrage, možemo rešiti simulacijom svih stanja, odnosno obilaskom celokupnog prostora pretrage **(eng. backtracking)**. Ovo je naravno korektan pristup, međutim on nije upotrebljiv za probleme sa velikim brojem stanja. Takođe, možemo koristiti **grabljivi pristup (eng. greedy algorithm)**, koji se zasniva na biranju lokalnog najboljeg poteza u svakoj tački grananja. Ova vrsta heuristike može znatno smanjiti složenost i prostor pretrage, ali se ne može tvrditi njena tačnost.

Ideja dinamičkog programiranja je da se iskoristi princip domina: sve nanizane domine će popadati ako se poruši prva domina u nizu. Dakle, da bi se rešio neki problem, treba rešiti neki njegov manji slučaj (podproblem), a zatim pokazati kako se rešenje zadatog problema može konstruisati polazeći od (rešenih) podproblema. Ovakav pristup je baziran na matematičkoj indukciji.

2 Šta je dinamičko programiranje?

Dinamičko programiranje je naziv popularne tehnike u programiranju kojom drastično možemo smanjiti složenost algoritma: od eksponencionalne do polinomijalne. Reč "programiranje" u samom nazivu tehnike se odnosi na tzv. tablični metod, a ne na samo kucanje kompjuterskog koda. Slično metodi **"podeli pa vladaj" (eng. divide and conquer)**, dinamičko programiranje rešavanje jednog problema svodi na rešavanje podproblema. Za ovakve probleme se kaže da imaju **optimalnu strukturu (eng. optimal substructure)**. "Podeli pa vladaj" algoritmi vrše particiju glavnog problema na nezavisne podprobleme. Zatim nastupa rekurzivno rešavanje podproblema, kako bi se njihovim spajanjem dobilo rešenje polaznog problema. Algoritam koji je dobar predstavnik ove klase jeste **sortiranje učešljavanjem (eng. merge sort)**, algoritam za sortiranje niza.

Dinamičko programiranje (DP) takođe vrši particiju glavnog problema na podprobleme **(eng. overlapping subproblems)**, ali koji nisu nezavisni. Još jedna bitna karakteristika DP jeste da se svaki podproblem rešava najviše jednom, čime se izbegava ponovno računanje numeričkih karakteristika istog stanja (odatle "tablični metod", iz prostog razloga što se, kako ćemo ubrzo videti, rešenja podproblema čuvaju u pomoćnim tablicama).

Opisanu memorizaciju objasnimo na primeru Fibonačijevih brojeva ($F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$ za $n > 2$). Naivna implementacija rekurzivne funkcije za računanje n-tog Fibonačijevog broja bazirana na definiciji je prikazana u algoritmu A.

Primetimo da pri računanju vrednosti $Fib(n)$, funkcije $Fib(m)$ za $m < n$ se pozivaju veći broj puta. Za $n = 5$ imali bi sledeće:

$$\begin{aligned} Fib(5) &= Fib(4) + Fib(3) \\ &= (Fib(3) + Fib(2)) + (Fib(2) + Fib(1)) \\ &= ((Fib(2) + Fib(1)) + Fib(2)) + (Fib(2) + Fib(1)) \end{aligned}$$

Algoritam A: Funkcija $Fib(n)$ za računanje n-tog Fibonačijevog broja

Input: Prirodni broj n
Output: n-ti Fibonačijev broj

```
1 if n ≤ 2 then
2     return 1;
3 end
4 return (Fib(n − 1) + Fib(n − 2));
```

Međutim ukoliko bi vršili memorizaciju izračunatih stanja, ne bi morali da "ulazimo" u rekurzije brojeva koje smo već računali. Definišimo niz $fib[k]$ u kome ćemo pamtiti Fibonačijeve brojeve koje računamo u rekurzivnim pozivima. Na početku inicijalizujmo sve elemente niza na -1 (čime ih praktično markiramo kao neizračunate). Ovim pristupom izbegavamo nepotrebna računanja.

Algoritam B: Funkcija $Fib(n)$ za računanje n-tog Fibonačijevog broja sa memorizacijom

Input: Prirodni broj n
Output: n-ti Fibonačijev broj

```
1 if fib[n] ≠ −1 then
2     return fib[n];
3 end
4 if n ≤ 2 then
5     fib[n] = 1;
6 end
7 fib[n] = Fib(n − 1) + Fib(n − 2);
8 return fib[n];
```

Grubo govoreći, strukturu algoritma baziranom na DP, možemo opisati u četiri koraka:

1. Okarakterisati strukturu optimalnog rešenja
2. Rekurzivno definisati optimalnu vrednost
3. Izračunati optimalnu vrednost problema "odzgo na gore"
4. Rekonstruisati optimalno rešenje

Kao rezultat prva tri koraka dobijamo optimalnu vrednost. Poslednji korak izvršavamo ukoliko je potrebno i samo optimalno rešenje. Pri njegovoj rekonstrukciji obično koristimo neke dodatne informacije koje smo računali u koraku 3.

Verovatno smo svi još u osnovnoj školi naišli na standardni problem kusura: prodavac ima diskretan skup novčanica sa vrednostima $1 = v_1 < v_2 < \ldots < v_n$ (beskonačno od svake vrste) i treba vratiti kusur u vrednosti od S novčanih jedinica, pri čemu treba minimizirati broj novčanica. Ukoliko je skup novčanica diskretan i menja se samo vrednost kusura S, problem možemo rešiti rekurzijom odnosno preko n ugnježdenih petlji. Međutim, šta se dešava u slučaju kada ni vrednosti novčanica, kao ni njihov broj nisu poznati unapred. Ovaj i mnoge druge probleme možemo jednostavno rešiti uz pomoć DP-a, dok je ovaj problem jedna verzija problema ranca koji ćemo obraditi u poglavlju 4.

Kako bi bolje razumeli koncept DP proći ćemo detaljno kroz naredni problem.

Problem 1 ***[Maksimalna suma nesusednih u nizu]*** *Dat je niz a prirodnih brojeva dužine n. Odrediti podniz datog niza čiji je zbir elemenata maksimalan, a u kome nema susednih elemenata.*

Podproblem gornjeg problema možemo definisati kao: nalaženje traženog podniza na nekom delu polaznog niza a. Preciznije, definisaćemo novi niz d na sledeći način:

$$d[k] = \text{maksimalan zbir nesusednih elemenata niza } (a_1, a_2, \ldots, a_k), \text{ za } k \in [1, \ldots, n]$$

Rešenje polaznog problema će biti vrednost $d[n]$.

Napomena: Algoritam za definisanja podproblema ne postoji i zavisi od same prirode problema. Nekada morate proći nekoliko mogućnosti kako bi došli do prave. Naravno, vežbom i detaljnijom analizom problema se može steći neka "intuicija", ali univerzalno rešenje ne postoji ("There is no free lunch").

Definišimo sada vrednosti niza d rekurzivno. Pretpostavimo da smo izračunali vrednosti niza d do $(k-1)$-og elementa i želimo izračunati k-ti. Posle dodavanja novog elementa a_k, za traženu optimalnu vrednost podniza $(a_1, a_2, \ldots, a_k)$ imamo dve mogućnosti: element a_k uključujemo u traženu sumu ili ne (elemente sa indeksima $1, \ldots, k-2$ koristimo u oba slučaja, jer na njih ne utiče činjenica da li je element k ušao u podniz ili ne). Ukoliko ga uključimo tada element na mestu $k-1$ ne sme ući u sumu (pošto je susedan sa k-tim); u suprotnom element a_{k-1} možemo uključiti. Ovo formalnije zapisujemo kao:

$$d[k] = \max\{d[k-1], a_k + d[k-2]\}, \text{ za } k \geq 3 \tag{1}$$

Definisanjem baze $d[1] = \max\{0, a_1\}$ i $d[2] = \max\{0, a_1, a_2\}$ vrednosti niza d možemo izračunati jednostavnom for ciklusom. Pri računanju vrednosti $d[n]$ preko formule (1), rekurzivnim pozivima bi mogli ići u beskonačnost ukoliko neke od vrednosti niza d nemamo izračunate bez rekurzije. U našem slučaju, elementi sa indeksima 1 i 2 se mogu inicijalizovati na gore opisan način. Dva uzastopna elementa niza d nam definišu bazu, zato što vrednost elementa sa indeksom n zavisi samo od prethodna dva elementa. Npr, u slučaju da je $d[k] = \max\{d[k-1], d[k-3]\} + 4$ bazu moraju da čine prva tri elementa niza d.

Napomena: Definisanje niza d je logička posledica gornje induktivne analize. Ispitujući slučajeve poslednjeg elementa niza dobili smo jednostavan uslov optimalne vrednosti.

Algoritam C: Pseudo kod problema maksimalne sume nesusednih u nizu

Input: Niz a prirodnih brojeva dužine n
Output: *subsequence* - podniz nesusednih elemenata sa najvećom sumom

```
1  if n = 1 then
2      return subsequence = a;
3  end
4  d[1] = max{0, a_1};
5  d[1] = max{0, a_1, a_2};
6  for k ← 3 to n do
7      if d[k − 1] > d[k − 2] + a[k] then
8          d[k] = d[k − 1];
9      else
10         d[k] = d[k − 2] + a[k];
11 end
12 subsequence = ∅;
13 currentIndex = n;
14 while (currentIndex > 0) do
15     if d[currentIndex] ≠ d[currentIndex − 1] then
16         add a[currentIndex] to subsequence;
17         currentIndex = currentIndex − 2;
18     else
19         currentIndex = currentIndex − 1;
20     end
21 end
22 return subsequence;
```

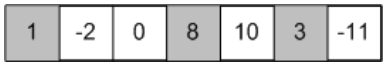
Opisanim algoritmom možemo izračunati maksimalnu sumu podniza nesusednih elemenata, ali ćemo se zadržati na rekonstrukciji samog podniza. U ovom konkretnom slučaju ne moramo pamtiti

dodatne informacije, pošto je rekurentna formula jednostavna. Za njegovu rekonstrukciju nam je potrebno da znamo da li je za vrednost $d[k]$ element na poziciji k ušao u podniz ili ne. Ukoliko bi element $a[n]$ pripadao traženom podnizu, tada bi vrednost $d[n]$ bila jednaka $d[n-2]+a[n]$. U slučaju da element sa indeksom n pripada, tada ispitujemo element sa indeksom $n-2$; u suprotnom element n ne pripada traženom podnizu, pa zato prelazimo na element $a[n-1]$. Treba napomenuti da opisanim postupkom dobijamo optimalni podniz u obrnutom porektu.

Složenost opisanog algoritma je linearna, $O(n)$ (računanje niz d je linearna, dok je rekonstrukcija traženog podniza složenosti $O(|subsequence|) = O(n)$). Uklapanjem svih koraka, algoritam možemo opisati pseudokodom u Algoritmu C.

Na primer, za dati niz $a = \{1, -2, 0, 8, 10, 3, -11\}$, niz d je

$$d = \{1, 1, 1, 9, 11, 12, 12\}$$



Slika 1: *Traženi podniz u primeru Problema 1*

Napomena: Opisani algoritam se može implementirati i rekurzivno. Parametar rekurzivne funkcije je index k i ona kao rezultat vraća vrednost $d[k]$. Kako bi izbegli ponovno računanje istih stanja, markiraćemo obiđena stanja i u slučaju da je stanje već računato, vratićemo odmah vrednost $d[k]$. Opšti opis algoritma dat je pseudo kodom u Algoritmu D.

Algoritam D: Opšti opis rekurzivne varijante DP-a

Input: stanje A
Output: optimalna vrednost stanja A, zapamćena u globalnom nizu d

```
1 if stanje A već obidjedno then
2     return d[A];
3 end
4 inicijalizuj vrednost d[A] rekurzivno;
5 markiraj stanje A kao obidjeno;
6 return d[A];
```

3 Poznatiji problemi

U ovom odeljku ćemo prodiskutovati neke poznatije predstavnike klase problema koje možemo rešiti uz pomoć DP. U primeru iz prethodnog dela smo videli kako možemo DP primeniti na jednodimenzionalnom problem, gde nam je podproblem bio okarakterisan jednim parametrom.

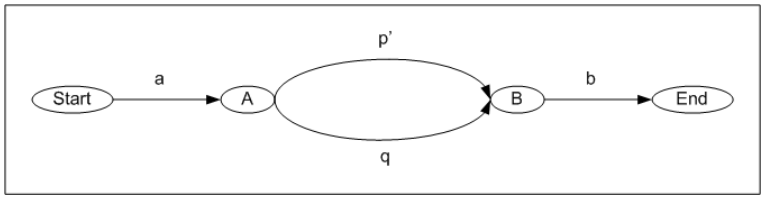
Problem 2 ***[Problem maksimalnog zbira u matrici]*** *Data je matrica a dimenzije $n \times m$ popunjena celim brojevima. Sa svakog polja u matrici dozvoljeno je preći samo na polje ispod ili na polje desno od tog polja (ukoliko postoje). Potrebno je izabrati put od gornjeg levog polja (polja sa koordinatama $(1,1)$), do donjeg desnog polja (polja sa koordinatama (n,m)), tako da zbir brojeva u poljima preko kojih se ide, bude maksimalan.*

Kao što se može pretpostaviti, generisanje svih puteva i pamćenje onog puta koji ima najveći zbir, nije dobra ideja. Broj mogućih puteva raste eksponencijalno sa porastom dimenzija matrice (preciznije, broj različitih puteva je jednak $\binom{n+m}{n}$).

Pokušajmo da razmišljamo unazad: neka je $P = \{p_1 = (1,1), \ldots, p_{n+m-1} = (n,m)\}$ traženi put sa maksimalnim zbirom. Kako se svakim potezom rastojanje do krajnjeg polja smanjuje za 1, znamo da je dužina traženog puta jednaka $n+m-1$. Tada svaki deo optimalnog puta P spaja polazno i završno

polje tog dela puta na optimalan način. Ovu činjenicu možemo jednostavno dokazati kontradikcijom: pretpostavimo da postoji optimalniji put koji spaja polja p_a i p_b od puta $p' = p_a, p_{a+1}, \ldots, p_b$. Označimo taj put sa q. Tada dobijamo da je put $p_1, p_2, \ldots, p_{a-1}, q, p_{b+1}, \ldots, p_{n+m-1}$ optimalniji od puta P, što je nemoguće. Na osnovu ove činjenice se prosto nameće definicija podproblema:

$$\begin{aligned} path[i][j] &= \text{optimalan put od polja } (1,1) \text{ do polja } (i,j) \\ d[i][j] &= \text{suma elemenata na optimalnom putu od polja } (1,1) \text{ do polja } (i,j) \end{aligned}$$



Slika 2: *Ukoliko je put $a - p' - b$ optimalan za stanja Start i End, tada je i put p' optimalan za stanja A i B*

Iz gornjih definicija imamo da će krajnji rezultat biti $d[n][m]$, odnosno $path[n][m]$. Pitanje koje se sada postavlja jeste da li se $d[i][j]$ može računati rekurzivno? Do polja (i,j) možemo doći preko polja $(i-1,j)$ i $(i,j-1)$ (pretpostavimo da polja postoje). Kako je svaki deo optimalnog puta optimalan, $d[i][j]$ definišemo kao:

$$d[i][j] = \max\{d[i-1][j], d[i][j-1]\} + a[i][j], \text{ za } i,j \geq 2$$

a tada se put $path[i][j]$ dobija dodavanjem polja (i,j) na onaj od puteva $path[i-1][j]$ ili $path[i][j-1]$ koji daje veći zbir u gornjoj jednačini.

Bazu DP će u ovom slučaju predstavljati elementi prve vrste i prve kolone matrice d pošto za njih ne važi gornja jednačina. Kako od polja $(1,1)$ do polja $(1,j)$ odnosno $(i,1)$ postoje jedinstveni putevi, popunjavanje matrice se može jednostavno realizovati na sledeći način:

$$d[1][1] = a[1][1],$$
$$d[1][j] = d[1][j-1] + a[1][j], \text{ za } j \geq 2,$$
$$d[i][1] = d[i-1][1] + a[i][1], \text{ za } i \geq 2,$$

što je jako slično rekurentnoj jednačini, stim što ovde nemamo biranje maksimuma iz dvoelementnog skupa – jer elementi imaju samo jednog suseda.

Kao i u prvom primeru, pri rekonstrukciji samog puta, nije potrebno pamtiti celokupnu matricu *path*. Traženi put se može pronaći kretanjem od nazad i biranjem onog polja (gornjeg ili levog) koje nam donosi veći zbir. Ukoliko memorijsko ograničenje to dozvoljava, možemo da pamtimo prethodno polje sa kojeg smo došli, čime bismo izbegli specijalne slučajeve koje nam skrivaju prva vrsta i kolona.

Put će opet biti u obrnutom redosledu – pa se ili mora rotirati na kraju ili pamtiti u steku. Međutim, kako u ovom slučaju znamo dužinu puta (u prvom problemu nismo znali kolika je dužina traženog podniza) možemo odmah i sam niz popunjavati od nazad, pa nećemo imati potrebe za njegovim rotiranjem.

Napomena: Problem sa specijalnih slučajeva prve vrste i kolone možemo rešiti elegantnije. Naime, problem rekurentne fromule je taj što navedena polja ne moraju uvek postojati. Ukoliko bi dodali još po jednu vrstu i kolonu sa indeksom 0 u matrici d i popunimo ih brojevima koji su sigurno manji od brojeva iz matrice, time obezbeđujemo da put forsirano ostane u prvobitnoj matrici. U ovom slučaju bazu DP ne bismo imali, tj. redovi $3, 4, \ldots, 10$ pseudo koda bi bila zamenjeni inicijalizacijom nulte vrste i kolone brojevima koji su manji od svih brojeva u matrici (koja su obično data kao ograničenja samog problema).

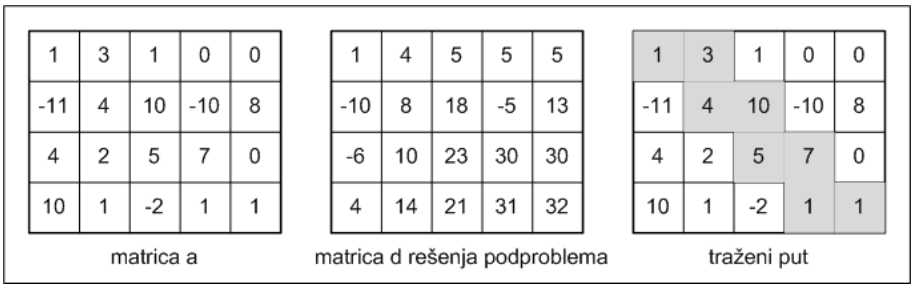
Na Slici 2 je prikazan rezultat algoritama na jednom primeru.

Ovaj problem se može sresti u više varijanti:

Algoritam E: Pseudo kod problema maksimalnog zbira u matrici

Input: Matrica a celih brojeva dimenzije $n \times m$
Output: *path* - niz elemenata matrice koji predstavlja traženi put

```
1  d[1][1] = a[1][1];
2  prior[1][1] = (0, 0);
3  for j ← 2 to m do
4      d[1][j] = d[1][j − 1] + a[1][j];
5      prior[1][j] = (1, j − 1);
6  end
7  for i ← 2 to n do
8      d[i][1] = d[i − 1][1] + a[i][1];
9      prior[i][1] = (i − 1, 1);
10 end
11 for i ← 2 to n do
12     for j ← 2 to m do
13         if d[i − 1][j] > d[i][j − 1] then
14             d[i][j] = d[i − 1][j] + a[i][j];
15             prior[i][j] = (i − 1, j);
16         else
17             d[i][j] = d[i][j − 1] + a[i][j];
18             prior[i][j] = (i, j − 1);
19         end
20     end
21 end
22 path = ∅;
23 currentField = (n, m);
24 while (currentField ≠ (0, 0)) do
25     add currentField in path;
26     currentField = prior[currentField.i][currentField.j];
27 end
28 return path;
```



Slika 3: *Primer problema maksimanog zbira u matrici*

- Prva varijanta se razlikuje u dodatnom uslovu: traženi put mora proći kroz polje (x, y). Ovo je jedan od čestih uslova DP problema, kada (uz uslov optimalnosti) rešenje mora proći kroz neka stanja. Rešenje ovakvog problema se obično svodi na početni problem koji nezavisno pozivamo nad delom ulaza. Konkretno u našem slučaju, traženi put će biti unija optimalnih puteva p i q, gde je p put od polja $(1, 1)$ do polja (x, y), a put q od polja (x, y) do polja (n, m).
- Druga varijanta dozvoljava da se iz jednog polja pređe ne samo u polja koja su desno i dole, već i u polje iznad. Naravno, ovde se mora dodati i uslov da se svako polje u putu obiđe najviše jednom (u suprotnom bi moglo doći do beskonačnog puta). Ovaj problem je komplikovaniji od prethodnih i nećemo ga ovde razmatrati.

Pre nego što pređemo na naredni problem, definišimo pojam podniza: Niz a je podniz niza b ukoliko se niz a može dobiti izbacivanjem nekih elemenata niza b. Na primer, imamo da je $\{2, 1, 7, 7\}$ podniza niza $\{5, 2, 1, 3, 7, 1, 7\}$, ali nije podniz od $\{2, 5, 7, 7, 1\}$.

Problem 3 ***[Najduži zajednički podniz (NZP)]*** *Data su dva niza a i b. Naći niz najveće moguće dužine koji je podniz i za a i za b.*

Problem najdužeg zajedničkog podniza **(eng. Longest Common Subsequence)** jako dobro demonstrira moć DP. Na početku, veoma brzo možemo uočiti da se uključivanjem bilo kakvih karakteristika problema ne može izbeći eksponencialna složenost, ukoliko problem ne posmatramo kroz podprobleme. Najlakše možemo baratati podnizovima koji su sastavljeni od početnih elemenata niza, pa zato matricu lcs definišemo kao:

$$lcs[x][y] = \text{NZP nizova } \{a_1, \ldots, a_x\} \text{ i } \{b_1, \ldots, b_y\}, \text{ za } 1 \leq x \leq n,\ 1 \leq y \leq m$$

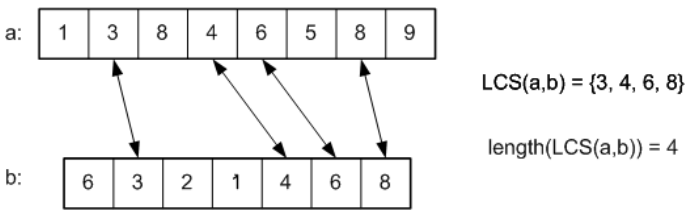
gde su n i m dužine nizova a odnosno b. U nastavku ćemo podniz $\{a_1, \ldots, a_x\}$ niza a označavati sa a^x. Pokušajmo sada da izrazimo vezu među elementima opisane matrice. Na početku, NZP za podnizove a^1 i b^k je $\{a_1\}$ ukoliko b^k sadrži element $a^1 \equiv a_1$; inače je $\emptyset$. Naravno, za sve elemente matrice važi $length(lcs[x][y]) \leq \min\{x, y\}$.

Posmatrajmo sada podnizove a^x i b^y. Za njihov NZP postoje dve mogućnosti: element $a[x]$ ulazi u NZP ili ne. Ukoliko ne ulazi, tada jednostavno imamo da je $lcs[x][y] = lcs[x-1][y]$. Pretpostavimo sada da $a[x]$ ulazi u NZP – tada podniz b^y mora sadržati element $a[x]$. Neka se element $a[x]$ nalazi na pozicijama $1 \leq c_1 < \ldots < c_k \leq y$ u podnizu b^y. Kako element $a[x]$ mora imati svog parnjaka u nizu b^y, imamo da je $lcs[x][y] = \max\{d[x-1][c_1 - 1], \ldots, d[x-1][c_k - 1]\} + \{a_x\}$. Kako je $length(lcs[x][y_1]) \leq length(lcs[x][y_2])$ za $y_1 \leq y_2$ prethodnu formulu možemo napisati kao

$$lcs[x][y] = lcs[x-1][c_k - 1] + \{a_x\}$$

gde je c_k index poslednjeg pojavljivanja elementa a_x u podnizu b^y. Kada složimo prethodnu priču, $lcs[x][y]$ se računa na sledeći način:

$$lcs[x][y] = \max\{lcs[x-1][y], lcs[x-1][c_k - 1] + \{a_x\}\} \tag{2}$$



Slika 4: *Primer najdužeg zajedničkog podniza*

Algoritam F: Pseudo kod problema NZP

Input: Dva niza a i b dužina n odnosno m
Output: LCS - najduži zajednički podniz

```
1  for j ← 0 to m do
2      lenLCS[0][j] = 0;
3  end
4  for i ← 0 to n do
5      lenLCS[i][0] = 0;
6  end
7  for i ← 1 to n do
8      for j ← 1 to m do
9          if a[i] = b[j] then
10             lenLCS[i][j] = lenLCS[i − 1][j − 1] + 1;
11         else
12             lenLCS[i][j] = max{lenLCS[i][j − 1], lenLCS[i − 1][j]};
13         end
14     end
15 end
16 LCS = ∅;
17 x = n; y = m;
18 while (x > 0 and y > 0) do
19     if a[x] = b[y] then
20         add a[x] to LCS;
21         x = x − 1; y = y − 1 ;
22     else
23         if lenLCS[x][y − 1] > lenLCS[x − 1][y] then
24             y = y − 1;
25         else
26             x = x − 1;
27         end
28     end
29 end
30 return LCS;
```

Pokušajmo da opišemo složenost gorenjeg algoritma. Kako jedini "problem" u formuli predstavlja pronalaženje samog indeksa c_k, možemo napisati da je složenost $O(n \times m \times FindC_k)$. Index c_k možemo pretraživati linearno što bi nas dovelo do složenosti $O(nm^2)$.

Ostavimo sada formulu (2) sa strane i pokušajmo da pristupimo računanju $lcs[x][y]$ na drugi način. Ukoliko bi elementi a_x i b_y bili jednaki, tada važi $lcs[x][y] = lcs[x-1][y-1] + \{a_x\}$. Šta bi bilo u slučaju da su oni različiti? Kako se navedeni elementi nalaze na poslednjim pozicijama nizova a^x i b^y, u slučaju da oba ulaze u NZP, tada bi morali biti jednaki, što je suprotno pretpostavci. Dakle, u tom slučaju možemo tvrditi da je $lcs[x][y] = \max\{lcs[x][y-1], lcs[x-1][y]\}$.

Gornja relacija nam omogućava da izračunamo sve elemente $lcs[x][y]$ redom po vrstama ili po kolonama, znajući da je $lcs[x][0] = lcs[0][y] = \emptyset$. Naravno, nije potrebno pamtiti same podnizove. Dovoljno je pamtiti njihove dužine – označićemo ih sa $lenLCS[x][y]$. Rekonstukcija samog podniza se slično prethodnim problemima može realizovati iz matrice $lenLCS$ ili pamćenjem prethodnika.

Složenost novog algoritma je $O(nm)$. Ova dva algoritma nam demonstriraju jednu bitnu činjenicu. Naime, nije uvek toliko očigledno koju rekurzivnu vezu treba uočiti. Za istu definiciju podproblema možemo imati više veza, koje kao što vidimo, ne moraju imati iste složenosti. Ovde čak imamo da je bolji algoritam i daleko jednostavniji za implementaciju.

Napomena: Ukoliko bi se tražila samo dužina NZPa, tada možemo da uštedimo memorijski prostor. Naime, umesto cele matrice $lenLCS$ možemo da imamo samo dva niza, koji bi igrali ulogu vrste iz $lenLCS$ koja se trenutno formira i prethodne vrste. Ovo možemo uraditi pošto nam izračunavanje elementa $lenLCS[x][y]$ zavisi samo od elemenata u vrstama x i $x-1$.

Problem 4 ***[Najduži rastući podniz (NRP)]*** *Data je niz a dužine n. Treba odrediti najduži podniz ne obavezno uzastopnih elemenata datog niza, koji je rastući.*

Prikazaćemo tri različita pristupa rešavanju problema najdužeg rastućeg podniza **(eng. Longest Increasing Subsequence)**. U definiciji problema je navedeno da se traži rastući podniz, tako da ćemo mi u daljem tekstu pod rastućim, bez gubljenja opštosti, podrazumevati strogu nejednakost. Na primeru $a = \{1, 9, 3, 8, 11, 4, 5, 6, 4, 19, 7, 1, 7\}$ možemo videti da je najduži rastući podniz dužine 6 i to $\{1, 3, 4, 5, 6, 7\}$.

Najjednostavnije rešenje jeste da svedemo ovaj problem na već poznati problem nalaženja najdužeg zajedničkog podniza. Konstruišemo niz b, koji predstavlja sortirani niz a u vremenu $O(n \log n)$, a zatim nađemo najduži zajednički podniz za nizove a i b u vremenu $O(n^2)$.

Takođe, postoji pravolinijsko rešenje dinamičkim programiranjem koje takođe radi u kvadratnoj složenosti. Iako su složenosti ekvivalentne, ovaj algoritam je znatno brži u praksi. Neka je $d[k]$ dužina najdužeg rastućeg podniza niza a, sa ograničenjem da je poslednji element upravo $a[k]$. Globalni NRP se mora završiti na nekom elementu niza a, pa ćemo konačno rešenje dobiti nalaženjem maksimuma u nizu d.

Ostaje da rekurzivno izračunamo elemente niza d. Kada računamo $d[k]$, posmatramo skup svih indeksa S_k za koje važi $i < k$ i $a[i] < a[k]$. Ako je skup S_k prazan, tada su svi elementi koji se nalaze pre $a[k]$ u nizu a veći od njega, što dovodi do $d[k] = 1$. Inače, ako maksimiziramo dužinu najdužeg rastućeg podniza u okviru skupa S_k, tada samo dodamo element $a[k]$ na kraj ovog niza. Zaključujemo da važi sledeća formula:

$$d[k] = \max\{d[i] \mid i \in S_k\} + 1 = \max_{i<k, a[i]<a[k]} d[i] + 1.$$

Ukoliko je potrebno nalaženje jednog NRP (pošto traženi podzniz ne mora biti jedinstven), to možemo uraditi korišćenjem pomoćnog niza p. Naime, $p[k]$ će nam predstavljati indeks i, koji je maksimizirao gornji izraz pri računanju $d[k]$. Takođe, najduži rastući podniz se može rekonstruisati i jednim prolaskom kroz niz od nazad.

Sada ćemo prikazati rešenje u vremenskoj složenosti $O(n \cdot \log k)$, gde je k dužina najdužeg rastućeg podniza. Definišimo $A_{i,j}$ kao:

Algoritam G: Pseudo kod problema NRP

Input: Niz a dužine n
Output: LIS - najduži rastući podniz niz a

```
1  for k ← 1 to n do
2      max = 0;
3      for i ← 1 to k − 1 do
4          if a[k] > a[i] and d[i] > max then
5              max = d[j];
6          end
7      end
8      d[k] = max + 1;
9  end
10 max = 0;
11 index = 0;
12 for k ← 1 to n do
13     if d[k] > max then
14         max = d[k];
15         index = k;
16     end
17 end
18 LIS ← ∅;
19 while d[index] > 1 do
20     add a[index] to LIS;
21     i = index − 1;
22     while d[index] > d[i] + 1 do
23         i = i − 1;
24     end
25     index = i;
26 end
27 add a[index] to LIS;
28 return LIS;
```

$A_{i,j}$ = najmanji mogući poslednji element svih rastućih nizova dužine j koristeći elemente $a[1], a[2], \ldots, a[i]$

Primetimo da za svako i, važi

$$A_{i,1} < A_{i,2} < \ldots < A_{i,j}.$$

Dakle, kada tražimo najduži rastući podniz koji se završava elementom $a[i+1]$ potrebno je da nađemo indeks j takav da je $A_{i,j} < a[i+1] \leqslant A_{i,j+1}$. U tom slučaju će tražena dužina biti jednaka $j+1$ i $A_{i+1,j+1}$ će biti jednako a_{i+1} dok će ostali elementi niza A_{i+1} ostati nepromenjeni. Zaključujemo da postoji najviše jedna razlika između nizova A_i i A_{i+1}. Kako je niz A_i uvek sortiran i posle primene gornje operacije - možemo koristiti **binarnu pretragu (eng. binary search)**.

Algoritam H: Pseudo kod problema NRP

Input: Niz a dužine n
Output: LIS - najduži rastući podniz

```
1  max = 1;
2  LIS[1] = 0;
3  for i ← 1 to n do
4      if a[LIS[max]] < a[i] then
5          p[i] = LIS[max];
6          max = max + 1;
7          LIS[max] = i;
8      end
9      else
10         left = 0;
11         right = max − 1;
12         while (left < right) do
13             m = (left + right)/2;
14             if a[LIS[m]] < a[i] then
15                 left = m + 1;
16             else
17                 right = m;
18             end
19         end
20         if a[i] < a[LIS[left]] then
21             if left > 0 then
22                 p[i] = LIS[left − 1];
23             end
24             LIS[left] = i;
25         end
26     end
27 end
28 i = LIS[max];
29 for k ← max to 1 do
30     i = p[i];
31     LIS[k] = i;
32 end
33 return LIS;
```

4 Problem ranca

Problem ranca (eng. Knapsack problem) je jedan od najpoznatijih problema DP. Ovaj problem takođe ima nekoliko varijanti. Jedna od jednostavnijih varijanti problema ranca je naredni problem, koji će nas lepo uvesti u suštinu osnovne formulacije problema.

Problem 5 *Dat je niz prirodnih brojeva a dužine n i prirodni broj $S \leq 10^5$. Pronaći podniz niza a čija je suma jednaka S ili ustanoviti da takav podniz ne postoji.*

Vidimo da u postavci problema postoji ograničenje za broj S. Kao što ćemo uskoro videti, ovo nam je potrebno pošto ćemo pratiti neka međustanja kojih će biti upravo S. Za početak pokušajmo da ustanovimo postojanje podniza, a potom ćemo videti kako ga i pronaći. Niz a dužine n ima tačno 2^n podnizova – pa ispitivanje svih podnizova odmah odbacujemo kao neefikasno rešenje. Definišimo zato niz sum dužine S na sledeći način:

$$sum[k] = \begin{cases} true, & \text{ukoliko postoji podniz sa sumom k} \\ false, & \text{inače} \end{cases}$$

Niz sum ćemo računati rekurzivno (dodavaćemo jedan po jedan element niza a i ažurirati vrednosti niza sum):

- Na početku sve elementa niza sum inicijalizujemo na $false$, osim $sum[0]$ koji će biti $true$. Ovo odgovara slučaju kada iz niza a nismo uzeli ni jedan element.
- Pretpostavimo da smo do sada dodali elemente $a[1], \ldots, a[k-1]$ (tj. podniz a^{k-1}) i želimo da dodamo element sa indeksom k. Koje elemente niza sum treba promeniti? Ukoliko je postojao podniz niza a^{k-1} koji je imao suma s, tada je taj isti podniz i podniz niza a^k – pa vrednost $sum[k]$ treba ostati nepromenjena i jednaka $true$. Međutim, dodavanjem novog elementa postoji i podniz čija je suma $s + a[k]$ koju treba postaviti na $true$ (ukoliko je ona jednaka $false$ i važi $s + a[k] \leq S$).

Odgovor na pitanje da li postoji podniz sa sumom S naći ćemo u vrednosti elementa $sum[S]$. Ukoliko nas zanima i sam podniz, pamtićemo za svaki element $sum[k]$ preko kog elementa niza a smo došli. Tačnije, uvek kada neki element niza sum postavimo na $true$, element iz druge stavke $a[k]$ pamtimo kao prethodnika. Kao prethodnika elementa $sum[0]$ kao i sve ostale do koji nismo uspeli da dođemo, postavićemo na -1. Na taj način, jednostavnom iteracijom po prethodnicima elementa $sum[S]$, sve dok taj prethodnik ne postane -1, dobijamo traženi podniz.

Napomena: Lako možemo primetiti da navedeni podniz nije jedinstven. Ukoliko bi želeli da rekonstruišemo sve podnizove, treba pamtiti sve prethodnike ili ih računati pri samoj rekonstrukciji (k-ti element je prethodnik sa element $sum[s]$ ukoliko je $sum[s - a[k]] = true$).

Napomena: Posebno pažnju treba obratiti na korak 7 u Algoritmu I, gde smo niz sum obilazili od nazad. U suprotnom bi imali slučaj da svaki element niza možemo upotrebiti proizvoljan broj puta. Naime, ukoliko bi niz a bio sastavljen samo od jednog elementa, naš niz sum bio imao vrednost $true$ na pozicijama $0, a[0], 2a[0], \ldots$. Razlog za ovo je jednostavan: $sum[a[0]]$ ćemo markirati preko $sum[0]$. Kasnijom iteracijom dolazimo na element $sum[a[0]]$ koji je $true$ zbog čega markiramo element $sum[a[0] + a[0]]$ itd.

Kao što smo napomenuli, problem ranca ima nekoliko varijanti. Dajemo onu formulaciju po kojoj je problem i dobio ime.

Problem 6 *Provalnik sa rancem zapremine N upao je u prostoriju u kojoj se čuvaju vredni predmeti. U prostoriji ima ukupno M predmeta. Za svaki predmet poznata je njegova vrednost $v[k]$ i njegova zapremina $z[k]$, $k \in [1, M]$. Sve navedene vrednosti su celobrojne. Provalnik želi da napuni ranac najvrednijim sadržajem. Potrebno je odrediti koje predmete treba staviti u ranac.*

Algoritam I: Pseudo kod problema podniza zadate sume

Input: Niz a dužine n; suma S koju ispitujemo
Output: *subArray* - podniz sa sumom S (prazan ukoliko takav ne postoji)

```
1  for i ← 0 to S do
2      sum[i] = false;
3      prior[i] = −1;
4  end
5  sum[0] = true;
6  for k ← 1 to n do
7      for s ← S downTo 0 do
8          if sum[s] and sum[s] + a[k] ≤ S then
9              sum[s + a[k]] = true;
10             prior[s + a[k]] = k;
11         end
12     end
13 end
14 subArray = ∅;
15 if sum[S] then
16     currentSum = S;
17     while prior[currentSum] ≠ −1 do
18         add prior[currentSum] to subArray;
19         currentSum = currentSum − a[prior[currentSum]];
20     end
21 end
22 return subArray;
```

Dakle, treba izabrati one predmete za koje je suma zapremina manja ili jednaka N, a čija je suma vrednosti maksimalna. Na početku prokomentarišimo nekoliko karakteristika problema:

- Ranac koji je optimalno popunjen, ne mora biti popunjen do vrha (u nekim slučajevima to i neće biti ni moguće). Na primer, posmatrajmo slučaj u kome je $N = 7$ a $v = \{3, 4, 8\}, z = \{3, 4, 5\}$. Ovde je optimalno rešenje uzeti samo poslednji predmet čime bi vrednost ranca bila 8, dok ranac ne bi bio u potpunosti popunjen.

- Najvredniji predmet ne mora ući rešenje: ideja da predmete treba sortirati po "vrednosti po jedinici zapremine", tj. po $v[k]/z[k]$ nije korektna. Ovaj pristup (grabljivi metod) ne daje uvek optimalno rešenje, što pokazuje primer: $N = 7$ i $v = \{3, 4, 6\}, z = \{3, 4, 5\}$. Ovde bi grabljivim algoritmom kao rešenje uzeli 3. predmet, dok se optimalno rešenje predstavljaju predmeti 1 i 2.

Iz ove diskusije se vidi da problem nije jednostavan i da se do rešenja ne može doći direktno. Analizirajući problem, možemo primetiti sledeće: ako je pri optimalnom popunjavanju ranca poslednji izabrani predmet k, onda preostali predmeti predstavljaju optimalno popunjavanje ranca zapremine $N - z[k]$. Ova konstatacija se lako dokazuje svođenjem na kontradikciju (kao i kod većine obrađenih problema). Prema tome, optimalno popunjavanje ranca sadrži optimalno popunjavanje manjeg ranca, što nas navodi na DP. Definišimo niz $d[k]$ za $k \in [1, N]$ kao:

$$d[k] = \text{maksimalna vrednost ranca zapremine } k \text{ pri čemu je ranac popunjen do vrha}$$

Optimalna vrednost ranca zapremine N se nalazi u nekom od elemenata $d[k]$ za $k \in [1, N]$. Iz gornjeg razmatranja vidimo da se računanje vrednosti $d[m]$ može svesti na računanje elementa $d[m - z[k]]$ za neki predmet k. Slično kao i kod prethodnog problema, niz d možemo računati dodavanjem jednog po jednog predmeta.

Na početku imamo slučaj praznog ranca i $d[0] = 0$ kao bazu. Kako se traži maksimalna vrednost ranca, možemo sve vrednosti niza d, osim pozicije 0, postaviti na $-\infty$ (na kraju algoritma ukoliko je

$d[k] = -\infty$ to će značiti da datim predmetima ne možemo popuniti u potpunosti ranac zapremine k). Pretpostavimo sada da smo dodali prvih $k-1$ predmeta i želimo da dodamo i k-ti predmet. Pitamo se šta bi bilo da ranac sadrži predmet k. Ukoliko bi sa njim ranac imao zapreminu $S \leq N$ tada bi ranac zapremine $S - z[k]$ bio popunjen nekim podskupom prvih $k-1$ predmeta. Zato ćemo svaki element $d[s] \neq -\infty$ pokušati da proširimo za novi predmet. Dakle, ispitujemo da li je $d[s + z[k]] < d[s] + v[k]$ i ukoliko jeste menjamo vrednost $d[s + z[k]]$ na optimalniju vrednost za ovu zapreminu – $d[s] + v[k]$.

Da bi rekonstruisali sam niz predmeta koje ćemo stavljati u ranac, za svaku vrednost $d[k]$ ćemo pamtiti indeks poslednjeg dodatog predmeta (u nizu *prior*). Pseudo kod gore opisanog algoritma je prikazan u Algoritmu J.

Algoritam J: Pseudo kod problema ranca

Input: Nizovi z i v dužine M; zapremina ranca N
Output: *knapsack* - niz optimalnih predmeta

```
1  for i ← 1 to S do
2      d[i] = −∞;
3      prior[i] = −1;
4  end
5  d[0] = 0;
6  for k ← 1 to M do
7      for i ← N − z[k] downTo 0 do
8          if d[i] ≠ −∞ and d[i + z[k]] < d[i] + v[k] then
9              d[i + z[k]] = d[i] + v[k];
10             prior[i + z[k]] = k;
11         end
12     end
13 end
14 knapsack = ∅;
15 index = max{i|i ∈ [0, N] and d[i] ≠ −∞};
16 while prior[index] ≠ −1 do
17     currentItem = prior[index];
18     add currentItem to knapsack;
19     index = prior[index − z[currentItem]];
20 end
21 return knapsack;
```

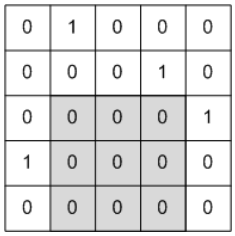
5 Razni zadaci

Ovde ćemo izneti nekoliko zadataka koji se rade metodom dinamičkog programiranja. Rešenja zadataka nisu obrađena do detalja sa pseudokodom kao u prethodnim, ali je opisan kompletan algoritam. Ukoliko ste prethodne probleme razumeli, onda neće biti problema oko implementacije algoritama u omiljenom programskom jeziku. Zadaci su birani tako da svaki zadatak krije po neku novu metodu DP koja se može koristiti u široj klasi problema.

Zadatak 5.1 (Srbija, Okružno takmičenje, 2000) *Data je binarna matrica dimenzije $n \times n$, $n \leq 1000$. Naći dimenziju najveće kvadratne podmatrice koja je sastavljena samo od nula.*

Rešenje: Svaku kvadratnu podmatricu možemo definisati njenom veličinom i jednim temenom, npr. donjim desnim. Pokušajmo da za svako polje matrice izračunamo najveću dimenziju podmatrice sastavljene samo od nula, tako da je dato polje upravo donji desni desni ugao. Označimo traženu veličinu sa $d[x][y]$.

Ukoliko bi $d[x][y]$ bilo jednako k, tada bi $d[x-1][y]$ moralo biti najmanje $k-1$, baš kao i $d[x][y-1]$. Međutim ovo nije i dovoljan uslov, jer nam on ništa ne govori o vrednosti u gornjem levom polju naše



Slika 5: *Primer najveće kvadratne podmatrice čiji su elementi samo 0*

podmatrice dimenzije k. Taj uslov možemo proveriti preko $d[x-1][y-1] \geq k-1$. Dakle, dobijamo

$$d[x][y] = \begin{cases} 0, & \text{ukoliko je } matrix[x][y] = 1 \\ 1 + \min\{d[x-1][y], d[x][y-1], d[x-1][y-1]\}, & \text{ukoliko je } matrix[x][y] = 0 \end{cases}$$

gde naravno moramo paziti da navedeni indeksi pripadaju matrici. Na početku možemo izračunati vrednosti u prvoj vrsti i koloni ili dodatni veštačke 0-te vrste i kolone. Ukoliko elemente matrice d računamo kretanjem po vrstama sa leva na desno u trenutku računanja elementa na poziciji (x, y) potrebne vrednosti sa desne strane jednakosti će biti već izračunate.

Zadatak 5.2 (TopCoder, SRM 255) *Dati su brojevi* $0 \leqslant M < N \leqslant 1.000.000$. *Odrediti najmanji broj sastavljen samo od neparnih cifara, koji pri deljenju sa* N *daje ostatak* M. *Ukoliko traženi broj ne postoji, štampati* -1.

Rešenje: Kako konstruisati sve brojeve sastavljene od neparnih cifara? Postoji tačno pet takvih jednocifrenih brojeva: 1, 3, 5, 7 i 9. Da bi dobili brojeve sa L neparnih cifara, dovoljno je pomnožiti svaki od $(L-1)$–cifrenih brojeva sastavljenih od neparnih cifara, sa 10 i dodati svaku od neparnih cifara.

Za rešenje će nam biti potreban jedan red q. Na početku ubacimo sve jednocifrene neparne brojeve u q. Zatim dodajemo sve neparnocifrene brojeve koji su dobijeni od 1, zatim sve neparnocifrene brojeve dobijene od 3 i tako dalje. Time ćemo konstruisati sve neparnocifrene brojeve u rastućem redosledu. Naravno, sve brojeve uzimamo po modulu N.

Nema smisla imati dva broja sa jednakim ostatkom u nizu q, pošto će oni generisati iste neparnocifrene brojeve. Zato koristimo niz d dužine N. Naime, $d[i]$ predstavlja najmanji broj koji pri deljenju sa N daje ostatak i. Na početku, inicijalizujemo elemente niza d na -1. Možemo primetiti da ne moramo pamtiti cele brojeve (pošto mogu biti sastavljeni od mnogo cifara), već je dovoljno pamtiti poslednju cifru i prethodnika.

Memorijska i vremenska složenost ovog algoritma je $O(N)$.

Zadatak 5.3 (Srbija, Republičko takmičenje 2003) *Dat je niz različitih prirodnih brojeva dužine* n. *Nad nizom možete izvršavati dve operacije: proizvoljan element iz niza prebacite na njegov početak ili na njegov kraj. Odrediti minimalni broj operacija, tako da rezultujući niz bude rastući.*

Rešenje: Označimo dati niz sa a i pretpostavimo da smo pri simulaciji sortiranja pomerili elemente sa indeksima $1 \leq x_1 < \ldots < x_k \leq n$. Elemente koje nismo premestili zadržali su svoj poredak kao i na početku. Kako na kraju treba dobiti sortirani niz, poredak nepremeštanih elemenata mora biti sortiran. Ova činjenica nas navodi na to da treba pronaći najduži rastući podniz niza. Međutim, primetimo da se nijedan novi element ne može ubaciti između neka dva elementa koja nismo premeštali. Zato nalazimo najduži rastući podniz niza a uzastopnih elemenata (ne po indeksima već po vrednostima).

Slika 6: *Primer sortiranja u Zadatku 5.2*

Na početku, radi lakše manipulacije, možemo elemente niza preslikati u permutaciju brojeva od 1 do n. Niz d definišimo kao:

$$\begin{aligned} d[n] &= \text{naduži rastući podniz elemenata sa indeksima iz segmenta } [1,n] \\ &\quad \text{koji se završava } n\text{–tim elementom} \end{aligned}$$

Niz d možemo računati linearno, za razliku od najdužeg rastućeg podniza, jer ovde znamo koji element jedini može da bude prethodnik (prethodnik za $d[k]$ je element koji ima vrednost $a[k]-1$).

Zadatak 5.4 (ACM SPOJ, PERMUT1) *Data su dva prirodna broja $n \leq 1000$ i $k \leq 10000$. Naći broj permutacija brojeva od 1 do n koji imaju tačno k inverzija. Broj inverzija u permutaciji $(p_1, p_2, \ldots, p_n)$ je broj parova (i,j) takvih da je $p_i > p_j$ za $1 \leq i < j \leq n$.*

Rešenje: Stanje ćemo opisati sa dva parametra: dužinom permutacije i brojem inverzija. Dakle, označimo sa

$$d[i][j] = \text{broj permutacija dužine } i \text{ sa } j \text{ inverzija}$$

Posmatrajmo jedinicu u tim permutacijama i njenu ulogu u broju inverzija. Ukoliko se ona nalazi na prvom mestu, onda ona ne učestvuje ni u jednoj inverziji, pa tih permutacija ima $d[i-1][j]$ (jer je svejedno da li gledamo permutacije brojeva $(1,2,\ldots,n-1)$ ili $(2,3,\ldots,n)$). Ako se jedinica nalazi na drugom mestu, tada permutacija sa j inverzija ima $d[i-1][j-1]$. Na ovaj način dolazimo do rekurentne formule:

$$d[n][k] = \sum_{i=0}^{n-1} d[n-1, k-i]$$

Pokušajmo da navedena polja matrice d računamo na optimalniji način.

$$d[n][k] = d[n][k-1] + d[n-1][k] - d[n-1][k-n]$$

Gornju relaciju nije teško primetiti: ukoliko ispišete sume za elemente $d[n][k]$ i $d[n][k-1]$ videćete da su one jako slične (tačnije razlikuju se samo u dva sabiraka). Ovo dodatno smanjuje kako vremensku, tako i memorijsku složenost. Elemente matrice d računamo u $O(1)$, dakle vremenska složenost algoritma je $O(nk)$ dok je memorijska $O(n)$, jer pamtimo samo poslednja dva reda matrice. Za početne vrednosti je dovoljno uzeti $d[i][0] = 1$ i $d[0][j] = 0$. Napomenimo da treba voditi računa o tome da li je broj inverzija k veći od brojača i i broja elemenata n.

Zadatak 5.5 (Hrvatska, Izborno 2001) *Dat je niz d realnih brojeva dužine $n \leq 1000$, koji predstavljaju udaljenost noćnih svetiljki od početka ulice. Za svaku od svetiljki je data i njena potrošnja u jedinici vremena kada je upaljena. Perica se na početku nalazi na rastojanju start od početka ulice i želi da pogasi sve svetiljke u ulici. On se kreće brzinom od jednog metra u jedinici vremena. Kolika je minimalna potrošnja svetiljki koje one proizvedu - dok ih Perica sve ne pogasi?*

Rešenje: Označimo potrošnju k-te sijalice sa $w[k]$. Na početku radi lakšeg računanja sortirajmo sijalice po rastojanjima i transformišimo koordinate tako da se Perica nalazi na mestu sa koordinatom 0. Ubacimo i jednu "veštačku sijalicu" na koordinati 0. Definišimo sledeće oznake:

$$\begin{aligned}
L[x][y] &= \text{minimalna potrošnja potrebna da se pogase sijalice sa indeksima } x, x+1, \ldots, y \\
&\quad \text{i da se na kraju Perica nalazi kod sijalice } x \\
D[x][y] &= \text{minimalna potrošnja potrebna da se pogase sijalice sa indeksima } x, x+1, \ldots, y \\
&\quad \text{i da se na kraju Perica nalazi kod sijalice } y \\
T[x][y] &= \text{ukupna snaga koju troše sijalice sa indeksima } x, x+1, \ldots, y \\
T^c[x][y] &= \text{ukupna snaga koju troše sijalice sa indeksima } 1, \ldots, x-1, y+1, \ldots, n
\end{aligned}$$

Elementi pomoćnih matrice T i T^c će nam biti potrebne za računanje glavnih matrica L i D u kojima se zadrži i krajnje rešenje $\min\{L[1][n], D[1][n]\}$. Razlog za tako definisane veze jeste činjenica da ukoliko Perica pogasi sijalice a i b tada će pogasiti i sijalice sa indeksima između a i b.

Pogledajmo prvo kako možemo elegantno izračunati matricu T, pošto ćemo sličnu ideju kasnije koristiti za matrice D i L. Pomoću jednakosti $T[x][y] = w[x] + T[x+1][y]$, računanje neke vrednosti za segment $[x, y]$ dijamatra $y - x$ možemo da svedemo na računanje segmenta $[x+1, y]$ koji je dijamatra $y - x - 1$. Ukoliko bi elemente matrice T računali na takav način da su u trenutku računanja polja dijametra k, elementi sa manjim dijametrima inicijalizovani, gornju jednakost bi mogli implementirati i bez rekurzije. Ideja je da se matrica T popunjava dijagonalno.

Algoritam K: Pseudo kod računanja matrice T iz zadatka 5.4

Input: Niz potrošnja sijalica w dužine n
Output: matrica T

```
1  for k ← 1 to n do
2      T[k][k] = w[k];
3  end
4  for diameter ← 2 to n − 1 do
5      for x ← 1 to n − diameter do
6          y = x + diameter − 1;
7          T[x][y] = w[x] + T[x + 1][y];
8      end
9  end
10 return T;
```

Elemente matrice T^c računamo kao $T^c[x][y] = S - T[x][y]$ gde je $S = \sum_{k=1}^{n} w[k]$. Sada možemo definisati rekurentne veze za matrice L i D.

$$L[x][y] = \min \begin{cases} L[x+1][y] + T^c[x+1][y] \cdot (d[x+1] - d[x]), \\ D[x+1][y] + T^c[x+1][y] \cdot (d[y] - d[x]) \end{cases}$$

$$D[x][y] = \min \begin{cases} D[x][y-1] + T^c[x][y-1] \cdot (d[y] - d[y-1]), \\ L[x][y-1] + T^c[x][y-1] \cdot (d[y] - d[x]) \end{cases}$$

Objasnimo rekurentnu jednačinu za $L[x][y]$, pošto se veza za matricu D dobija analogno. Naime, da bi Perica pogasio sijalice iz segmenta $[x, y]$ na kraju mora biti ili kod sijalice x ili kod sijalice y. Kako bi ugasio sijalice $[x, y]$ mora pre toga ugasiti sijalice $[x-1, y]$. Za njihovo gašenje postoje dva slučaja: da se nalazi u $x - 1$ ili y, što ujedno i predstavljaju gornje slučajeve za računanje elementa $L[x][y]$.

Reference

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, The MIT Press, 2001

[2] Udi Manber, *Introduction to Algorithms*, Addison-Wesley, Boston, 1989

[3] Robert Sedgewick, *Algorithms*, Addison-Wesley, 1984

[4] Donald E. Knuth, *The Art of Computer Programming Volume 1,2,3*, Addison-Wesley, 1973

[5] Milan Vugdelija, *Dinamičko programiranje*, Beograd, 1999

[6] http://www.topcoder.com/tc/

[7] http://www.uwp.edu/sws/usaco/

[8] http://www.spoj.pl/

[9] http://acm.timus.ru/

[10] http://www.z-trening.com/

[11] http://acm.uva.es/problemset/

[12] http://www.dms.org.yu/