Čas 11.1, 11.2, 11.3 - dinamičko programiranje

Pojam i oblici dinamičkog programiranja

U mnogim slučajevima se dešava da tokom izvršavanja rekurzivne funkcije dolazi do *preklapanja rekurzivnih poziva* tj. da se identični rekurzivni pozivi izvršavaju više puta. Ako se to dešava često, programi su po pravilu veoma neefikasni (u mnogim slučajevima broj rekurzivnih poziva, pa samim tim i složenost biva eksponencijalna u odnosu na veličinu ulaza). Do efikasnijeg rešenja se često može doći tehnikom *dinamičkog programiranja*. Ono često vremensku efikasnost popravlja angažovanjem dodatne memorije u kojoj se beleže rezultati izvršenih rekurzivnih poziva. Dinamičko programiranje dolazi u dva oblika.

- Tehnika *memoizacije* ili *dinamičkog programiranja naniže* zadržava rekurzivnu definiciju ali u dodatnoj strukturi podataka (najčešće nizu ili matrici) beleži sve rezultate rekurzivnih poziva, da bih ih u narednim pozivima u kojima su parametri isti samo očitala iz te strukture.
- Tehnika *dinamičkog programiranja naviše* u potpunosti uklanja rekurziju i tu pomoćnu strukturu podataka popunjava iscrpno u nekom sistematičnom redosledu.

Dok se kod memoizacije može desiti da se rekurzivna funkcija ne poziva za neke vrednosti parametara, kod dinamičkog programiranja naviše se izračunavaju vrednosti funkcije za sve moguće vrednosti njenih parametara manjih od vrednosti koja se zapravo traži u zadatku. Iako se na osnovu ovoga može pomisliti da je memoizacija efikasnija tehnika, u praksi je češći slučaj da je tokom odmotavanja rekurzije potrebno izračunati vrednost rekurzivne funkcije za baš veliki broj različitih parametara, tako da se ova efikasnost u praksi retko sreće.

Najbolji način da razjasnimo tehniku dinamičkog programiranja je da je ilustrujemo na nizu pogodno odabranih primera. Krenućemo od Fibonačijevog niza, koji je opšte poznati problem i kroz čije se rešavanje mogu ilustrovati većina osnovnih koncepata dinamičkog programiranja.

Fibonačijevi brojevi

Problem: Napisati program koji za dato n izračunava F_n, gde je F_n Fibonačijev niz definisan pomoću $F_0 = 0$, $F_1 = 1$ i $F_n = F_{n-1} + F_{n-2}$, za $n > 1$.

Pošto je već sama definicija rekurzivna, funkcija se može implementirati krajnje jednostavno.

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Međutim, broj sabiranja koji se vrši tokom izvršavanja ove funkcije zadovoljava jednačinu $T(n) = T(n-1)+T(n-2)+1$ za $n > 1$ i $T(0) = T(1) = 0$. Rešenje ove nehomogene jednačine jednako je zbiru rešenja njenog homogenog dela (a to je upravo jednačina Fibonačijevog niza, čije rešenje raste eksponencijalno) i jednog njenog partikularnog rešenja, pa je jasno da je broj sabiranja eksponencijalan u odnosu na n. Uzrok tome je veliki broj preklapajućih rekurzivnih poziva. Ako funkciju modifikujemo tako da na početku svog izvršavanja ispisuje broj n, za poziv `fib(6)` dobijamo sledeći ispis.

```
6 5 4 3 2 1 0 1 2 1 0 3 2 1 0 1 4 3 2 1 0 1 2 1 0
```

Vrednost `4` se javila kao parametar 2 puta, vrednost `3` se javila kao parametar 3 puta, vrednost `2` se javila 5 puta, vrednost `1` se javila 8 puta, a vrednost `0` 5 puta. Primećujemo da ovo odgovara elementima Fibonačijevog niza. Rekurzivni pozivi se mogu predstaviti i drvetom.

```
                          6
               5                   4
        4           3          3        2
    3      2      2   1      2   1    1   0
  2   1  1   0  1   0      1   0
1   0
```

Veoma značajno ubrzanje se može dobiti ako upotrebimo tehniku memoizacije. Rezultate svih rekurzivnih poziva ćemo pamtiti u nekoj pomoćnoj strukturi podataka. Pošto vrednosti parametara treba da preslikamo u rezultate rekurzivnih poziva treba da koristimo neku rečničku strukturu. To može biti mapa, odnosno rečnik.

```
int fib(int n, map<int, int>& memo) {
  // ako smo već računali vrednost za parametar n
  // rezultat samo očitavamo iz mape
  auto it = memo.find(n);
  if (it != memo.end())
     return it->second;

  // pre nego što vratimo rezultat, pamtimo ga u mapi
  if (n == 0) return memo[n] = 0;
  if (n == 1) return memo[n] = 1;
  return memo[n] = fib(n-1) + fib(n-2);
}

int fib(int n) {
  map<int, int> memo;
```

```
  return fib(n, memo);
}
```

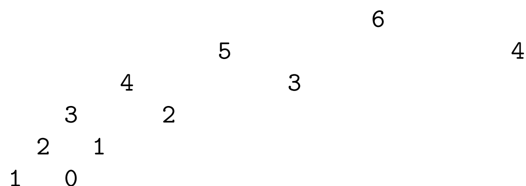
Iako mapa tj. rečnik predstavlja prirodan izbor za čuvanje konačnog preslikavanja, njena upotreba u službi memoizacije nije česta. Naime, pokazuje se da se značajno bolje performanse postižu ako se umesto mape upotrebi niz ili vektor. Time se može angažovati malo više memorije, međutim, pretraga vrednosti je značajno brža. U situacijama u kojima se vrednost izračunava za veliki broj ulaznih parametara (a kod Fibonačija možemo biti sigurni da se prilikom izračunavanja vrednosti za parametar n vrše pozivi za sve vrednosti o 0 do $n-1$), niz može biti čak i memorijski efikasniji u odnosu na mapu. Stoga ćemo nadalje u sklopu dinamičkog programiranja koristiti nizove tj. vektore.

Rečnik ćemo pamtiti preko niza tako što ćemo na mestu i pamtiti vrednost poziva za vrednost parametra i. Potrebno je još nekako obeležiti vrednosti parametara u nizu za koje još ne znamo rezultate rekurzivnih poziva. Za to se obično koristi neka specijalna vrednost. Ako znamo da će svi rezultati biti nenegativni brojevi, možemo upotrebiti, na primer, -1, a ako znamo da će biti pozitivni brojevi, možemo upotrebiti, na primer 0. Ako nemamo takvih pretpostavki možemo angažovati dodatni niz logičkih vrednosti kojima ćemo eksplicitno kodirati da li za neki parametar znamo ili ne znamo vrednost. Pošto smo sigurni da će tokom rekurzije sve vrednosti parametara pozitivne i da je najveća vrednost koja se može javiti kao parametar vrednost inicijalnog poziva n, dovoljno je da alociramo niz veličine $n+1$.

```
int fib(int n, vector<int>& memo) {
  // ako je vrednost za parametar n već računata
  // vraćamo ranije izračunatu vrednost
  if (memo[n] != -1)
     return memo[n];
  // pre nego što vratimo vrednost, pamtimo je u nizu
  if (n == 0) return memo[n] = 0;
  if (n == 1) return memo[n] = 1;
  return memo[n] = fib(n-1, memo) + fib(n-2, memo);
}

int fib(int n) {
  // alociramo niz veličine n+1 i popunjavamo ga vrednostima -1
  // kojima označavamo da ta vrednost još nije računata
  vector<int> memo(n+1, -1);
  return fib(n, memo);
}
```

Na ovaj način dobijamo algoritam čija je i vremenska i memorijska složenost $O(n)$, jer se za svako n izračunavanje vrši samo jednom. Drvo rekurzivnih poziva u ovom slučaju izgleda ovako (spustom duž leve grane računaju se vrednosti za sve parametre, dok se onda svaka od desnih grana saseca jer je rezultat već od ranije poznat).



Tehnika dinamičkog programiranja naviše podrazumeva da se ukloni rekurzija i da se sve vrednosti u nizu popune nekim redosledom. Pošto vrednosti na višim pozicijama zavise od onih na nižim, niz popunjavamo sleva nadesno. Na prva dva mesta upisujemo nulu i jedinicu, a zatim u petlji svaku narednu vrednost izračunavamo na osnovu dve prethodne. Na kraju vraćamo traženu vrednost na poslednjoj poziciji u nizu.

```
int fib(int n) {
   vector<int> dp(n+1);
   dp[0] = 0;
   if (n == 0) return dp[n];
   dp[1] = 1;
   if (n == 1) return dp[1];
   for (int i = 2; i <= n; i++)
     dp[i] = dp[i-1] + dp[i-2];
   return dp[n];
}
```

I u ovom slučaju je jasno da je složenost izračunavanja $O(n)$. Vreme izračunavanja F_{45} u slučaju obične rekurzivne funkcije je oko $3,627$ sekundi, dok je u obe varijante dinamičkog programiranja ono oko $0,005$ sekundi.

Pažljivijom analizom možemo ustanoviti da nam niz zapravo i nije potreban. Naime, svaka naredna vrednost zavisi tačno od dve prethodne. Svaka se vrednost koristi prilikom izračunavanja dve vrednosti iza nje. Jednom kada se one izračunaju, ta vrednost više nije potrebna. Zato je dovoljno u svakom trenutku pamtiti samo dve uzastopne vrednosti u nizu.

```
int fib(int n) {
   int pp = 0;
   if (n == 0) return pp;
   int p = 1;
   if (n == 1) return p;
   for (int i = 2; i <= n; i++) {
      // (pp, p) = (p, pp+p)
      int f = pp + p;
      pp = p;
      p = f;
   }
   return p;
}
```

Ovim smo izvršili redukciju memorijske složenosti i dobili algoritam čija je memorijska složenost umesto $O(n)$ jednaka $O(1)$.

Niz koraka koji smo primenili u ovom zadatku javljaće se veoma često.

1. Induktivno-rekurzivnom konstrukcijom konstruiše se rekurzivna definicija koja je neefikasna jer se isti pozivi prekapaju tj. funkcija se za iste argumente poziva više puta.
2. Tehnikom memoizacije poboljšava se složenost tako što se u pomoćnom rečniku (najčešće implementiranom pomoću niza ili matrice) čuvaju izračunati rezultati rekurzivnih poziva.
3. Umesto tehnike memoizacije koja je vođena rekurzijom i u kojoj se vrednosti popunjavaju po potrebi, rekurzija se eliminiše i rečnik (niz tj. matrica) se popunjava iscrpno nekim redosledom.
4. Vrši se memorijska optimizacija na osnovu toga što se primećuje da nakon popunjavanja određenih elemenata niza tj. matrice neke vrednosti (raniji elementi, ranije vrste ili kolone) više nisu potrebne, tako da se umesto istovremnog pamćenja svih elemenata pamti samo nekoliko prethodnih (oni koji su potrebni za dalje popunjavanje).

Prebrojavanje kombinatornih objekata

Jedan domen u kom se dinamičko programiranje često primenjuje je prebrojavanje kombinatornih objekata. O generisanju kombinatornih objekata već je bilo reči u poglavlju o pretrazi. U ovom poglavlju ćemo videti kako su generisanje i prebrojavanje zapravo tesno povezani (naravno, do algoritama prebrojavanja se može doći i nezavisno od rekurzivnog generisanja, ali su ideje koje se u pozadini kriju zapravo identične).

Broj kombinacija

Problem: Napiši program koji određuje broj kombinacija dužine k iz skupa od n elemenata.

Jedna interesantna tehnika koju smo već sreli prilikom empirijske analize složenosti funkcije QuickSort prilagođava algoritam tako da izračunava broj koraka koji se u njemu izvršavaju. Tako možemo krenuti od funkcije koju smo izveli za generisanje svih kombinacija.

```
void obradiSveKombinacije(vector<int>& kombinacija, int i,
                          int n_min, int n_max) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
```

```
    obradi(kombinacija);
    return;
  }

  // ako tekuću kombinaciju nije moguće popuniti do kraja
  // prekidamo ovaj pokušaj
  if (k - i > n_max - n_min + 1)
    return;

  // vrednost n_min uključujemo na poziciju i, pa rekurzivno
  // proširujemo tako dobijenu kombinaciju
  kombinacija[i] = n_min;
  obradiSveKombinacije(kombinacija, i+1, n_min+1, n_max);
  // vrednost n_min preskačemo i isključujemo iz kombinacije
  obradiSveKombinacije(kombinacija, i, n_min+1, n_max);
}
```

Funkcija treba da vraća broj kombinacija. Ako nam je bitan broj kombinacija, a ne i same kombinacije, tada možemo u potpunosti izbaciti iz igre niz koji se popunjava i umesto njega prosleđivati samo njegovu dužinu k.

```
int brojKombinacija(int i, int k,
                    int n_min, int n_max) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (i == k) return 1;
  // ako niz nije moguće popuniti do kraja, tada nema kombinacija
  if (k - i > n_max - n_min + 1)
    return 0;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k, i+1, n_min+1, n_max) +
         brojKombinacija(k, i, n_min+1, n_max);
}
```

Možemo primetiti da nam konkretne vrednosti k i i nisu bitne, već je bitan samo broj elemenata u intervalu $[i, k)$ tj. razlika $k - i$. Slično, nisu nam bitne ni konkretne vrednosti n_{max} i n_{min} već samo broj elemenata u segmentu $[n_{min}, n_{max}]$ tj. vrednost $n_{max} - n_{min} + 1$. Ako te dve veličine zamenimo sa k tj. n dobijamo narednu definiciju.

```
int brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako niz nije moguće popuniti do kraja, tada nema kombinacija
  if (k > n) return 0;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}
```

Ako funkciju pozovemo za vrednosti $k \leq n$, slučaj $k > n$ može nastupiti jedino iz drugog rekurzivnog poziva za $k = n$ (jer odnos između k i n u prvom rekurzivnom

pozivu ostaje nepromenjen, a u drugom se menja samo za 1). Međutim, u slučaju poziva funkcije za $k = n$ dobiće se uvek povratna vrednost 1 (drugi rekurzivni poziv će uvek vraćati nule, a prvi će prouzrokovati smanjivanje oba argumenta sve dok se ne dođe do $k = n = 0$, kada će se 1 vratiti na osnovu prvog izlaza iz rekurzije), što je sasvim u skladu sa tim da tada postoji samo jedna kombinacija. Na osnovu ovoga iz rekurzije možemo izaći za $k = n$ vrativši vrednost 1, čime onda eliminišemo potrebu za proverom da li je $k > n$ (naravno, pod pretpostavkom da ćemo funkciju pozivati samo za $k \leq n$).

```
int brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako treba popuniti još tačno n elemenata, tada postoji
  // tačno jedna kombinacija
  if (k == n) return 1;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}
```

Primećujemo da smo ovom transformacijom dobili čuvene osobine binomnih koeficijenata.

$$\binom{n}{0} = 1, \qquad \binom{n}{n} = 1, \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

One čine osnovu Paskalovog trougla u kom se nalaze binomni koeficijenti.

```
1                          (0,0)
1  1                       (1,0) (1,1)
1  2  1                    (2,0) (2,1) (2,2)
1  3  3  1                 (3,0) (3,1) (3,2) (3,3)
1  4  6  4  1              (4,0) (4,1) (4,2) (4,3) (4,4)
1  5 10 10  5  1           (5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
1  6 15 20 15  6  1        (6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6)
```

Prva veza govori da su elementi prve kolone uvek jednaki 1, druga da su na kraju svake vrste elementi takođe jednaki 1, a treća da je svaki element u trouglu jednak zbiru elementa neposredno iznad njega i elementa neposredno ispred tog.

Iako korektna, gornja funkcija je neefikasna i može se popraviti tehnikom dinamičkog programiranja. Najjednostavnije prilagođavanje je da se upotrebi memoizacija. Pošto funkcija ima dva parametra, za memoizaciju ćemo upotrebiti matricu. Ako se $\binom{n}{k}$ pamti u matrici na poziciji (n, k), matricu možemo alocirati na $n+1$ vrsta, gde poslednja vrsta ima $n+1$ elemenata, a svaka prethodna jedan element manje (u matricu će se popunjavati elementi Paskalovog trougla). Pošto nas neće zanimati vrednosti veće od polaznog k i pošto se i k i n smanjuju tokom rekurzije, pri čemu je $k \leq n$, možemo i odseći deo trougla desno od pozicije k.

Pošto su brojevi kombinacija uvek veći od nule, vrednosti 0 u matrici će nam označavati da poziv za te parametre još nije izvršen.

```
int brojKombinacija(int k, int n, vector<vector<int>>& memo) {
  // ako smo već računali broj kombinacija, ne računamo ga ponovo
  if (memo[n][k] != 0) return memo[n][k];

  // broj kombinacija na početku i na kraju svake vrste jednak je 1
  if (k == 0 || k == n) return memo[n][k] = 1;
  // broj kombinacija u sredini jednak je zbiru
  // broja kombinacija iznad i iznad levo od tekućeg elementa
  return memo[n][k] = brojKombinacija(k-1, n-1, memo) +
                      brojKombinacija(k, n-1, memo);
}

int brojKombinacija(int K, int N) {
  // alociramo prostor za rezultate rekurzivnih poziva koji se
  // mogu desiti i popunjavamo matricu nulama
  vector<vector<int>> memo(N+1);
  for (int n = 0; n <= N; n++)
    memo[n].resize(min(K+1, n+1), 0);
  // pozivamo funkciju koja će izračunati traženi broj
  return brojKombinacija(K, N, memo);
}
```

Umesto memoizacije možemo upotrebiti i dinamičko programiranje naviše, osloboditi se rekurzije i popuniti trougao vrstu po vrstu naniže. Popunjavanje celog trougla je prilično jednostavno.

```
int brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje celog trougla
  vector<vector<int>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(n+1);
  // obrađujemo vrstu po vrstu
  for (int n = 0; n <= N; n++) {
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata iznad njih
    for (int k = 1; k < n; k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // na kraju svake vrste nalazi se 1
    dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}
```

I u ovom slučaju možemo odseći nepotrebne desne kolone u trouglu.

```
int brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje relevantnog dela trougla
  vector<vector<int>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(min(K+1, n+1));
  // trougao popunjavamo kolonu po kolonu
  for (int n = 0; n <= N; n++) {
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata iznad njih
    for (int k = 1; k <= min(n-1, K); k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // ako je potrebno da znamo krajnji element kolone, postavljamo
    // ga na vrednost 1
    if (n <= K)
       dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}
```

Pažljivijom analizom prethodnog koda vidimo da, kako je to obično slučaj u dinamičkom programiranju, ne moramo istovremeno čuvati sve elemente matrice, jer svaka vrsta zavisi samo od prethodne i dovoljno je umesto matrice čuvati samo njene dve vrste (prethodnu i tekuću). Zapravo, dovoljno je čuvati samo jedan vektor vrstu ako je pažljivo popunjavamo i ako tokom njenog ažuriranja u jednom njenom delu čuvamo tekuću, a u drugom narednu vrstu. Pošto element (n, k) zavisi od elementa $(n-1, k-1)$ i od elementa $(n-1, k)$ znači da svaki element zavisi od elemenata koji su levo od njega, ali ne od elemenata koji su desno od njega. Zato ćemo vektor popunjavati zdesna nalevo. Pretpostavićemo da tokom ažuriranja važi invarijanta da se na pozicijama strogo većim od k nalaze elementi vrste n, a da se na pozicijama manjim ili jednakim od k nalaze elementi vrste $n-1$. Ažuriranje započinje time što na kraj vrste dopišemo vrednost 1 (osim u slučaju kada vršimo sasecanje desnog dela trougla) i nastavlja se tako što se element na poziciji k uveća za vrednost na poziciji $k-1$. Zaista, pre ažuriranja se na poziciji k nalazi vredost trougla sa pozicije $(n-1, k)$, dok se na poziciji $k-1$ nalazi vrednost trougla sa pozicije $(n-1, k-1)$. Njihov zbir je vrednost trougla na poziciji (n, k), pa se on upisuje na poziciju k i nakon toga se k smanjuje za 1, čime se invarijanta održava. Ažuriranje se vrši do pozicije $k = 1$, jer se na poziciji $k = 0$ u svim vrstama nalazi vrednost 1.

```
int brojKombinacija(int K, int N) {
  // tekuća vrsta
  vector<int> dp(K+1);
  // na početku svake vrste nalazi se 1
  dp[0] = 1;
  // trougao popunjavamo vrstu po vrstu
  for (int n = 1; n <= N; n++) {
```

```
    // vrstu ažruriramo zdesna nalevo
    // na kraju svake vrste nalazi se 1
    if (n <= K) dp[n] = 1;
    // ažuriramo unutrašnje elemente
    for (int k = min(n-1, K); k > 0; k--)
        dp[k] += dp[k-1];
  }
  // vraćamo traženi rezultat
  return dp[K];
}
```

Memorijska složenost ovog rešenja je $O(k)$, dok je vremenska $O(n \cdot k)$. Primetimo kako smo od veoma neefikasnog rešenja eksponencijalne složenosti tehnikom dinamičkog programiranja dobili veoma efikasno i uz to prilično jednostavno rešenje.

Recimo i da smo mogli krenuti od algoritma nabrajanja svih kombinacija u kom se u petlji razmatraju svi kandidati za element na tekućoj poziciji. Time bi se dobio algoritam koji bi element $\binom{n}{k}$ računao po sledećoj formuli:

$$\binom{n}{k} = \sum_{n'=k}^{n} \binom{n'}{k-1}$$

Broj kombinacija sa ponavljanjem

Razmotrimo i problem određivanja broja kombinacija sa ponavljanjem. Opet se jednostavnim transformacijama koda koji vrši njihovo generisanje može doći do naredne rekurzivne definicije. Naravno, do ovoga se može stići i direktnim razmatranjem. Naime, ako je potrebno izabrati 0 elemenata iz skupa od n elemenata to je moguće uraditi samo na jedan način. Ako je potrebno izabrati $k > 0$ elemenata iz praznog skupa, to nije moguće učiniti. U suprotnom, sve kombinacije možemo podeliti na one koje počinju najmanjim elementom skupa od n elemenata i na one koji ne počinju njime. Možemo dakle, uzeti najmanji element skupa i zatim gledati sve moguće načine da se odabere $k-1$ element iz istog n-točlanog skupa ili možemo svih k elemenata izabrati iz skupa iz kog je izbačen taj najmanji element.

```
int brojKombinacijaSaPonavljanjem(int k, int n) {
  if (k == 0) return 1;
  if (n == 0) return 0;
  return brojKombinacijaSaPonavljanjem(k-1, n) +
         brojKombinacijaSaPonavljanjem(k, n-1);
}
```

Broj ovih kombinacija može se rasporediti u pravougaonik.

```
0  1   1   1   1   1        (0,0) (0,1) (0,2) (0,3) (0,4) (0,5)
0  1   2   3   4   5        (1,0) (1,1) (1,2) (1,3) (1,4) (1,5)
0  1   3   6  10  15        (2,0) (2,1) (2,2) (2,3) (2,4) (2,5)
0  1   4  10  20  35        (3,0) (3,1) (3,2) (3,3) (3,4) (3,5)
0  1   5  15  35  70        (4,0) (4,1) (4,2) (4,3) (4,4) (4,5)
0  1   6  21  56 126        (5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
```

Razmotrimo optimizovano rešenje dinamičkim programiranjem. Vrsta za sledeće k se može dobiti ažuriranjem vrste za prethodno k. Pošto svaki element u pravougaoniku zavisi od vrednosti iznad i levo od sebe, vrstu možemo ažurirati sleva nadesno. Invarijanta je da se u trenutku ažuriranja pozicije n, na pozicijama strogo manjim od n nalaze vrednosti iz tekuće kolone k, a na pozicijama od n nadalje se nalaze vrednosti iz prethodne kolone $k-1$. Element na poziciji n koji sadrži vrednost sa pozicije $(k, n-1)$ pravougaonika uvećavamo za vrednost levo od njega koji sadrži vrednost $(k-1, n)$ i tako dobijamo vrednost elementa na poziciji (k, n). Uvećavanjem vrednosti n za 1 se održava invarijanta.

```
int brojKombinacijaSaPonavljanjem(int K, int N) {
  vector<int> dp(N+1, 1);
  dp[0] = 0;
  for (int k = 1; k <= K; k++)
     for (int n = 1; n <= N; n++)
        dp[n] += dp[n-1];
  return dp[N];
}
```

Postoji i veoma zgodan algoritam da se kombinacije sa ponavljanjem svedu na obične kombinacije. Zamislimo da smo poređali sve elemente od 1 do n i da pravimo program za robota koji će odabrati kombinaciju tako što kreće od prvog broja i u svakom trenutku ili može da uzme taj broj (operacija $+$) ili da se pomeri na sledeći broj (operacija $\rightarrow$), sve dok ne stigne do poslednjeg broja, pri čemu treba ukupno da uzme k brojeva. Na primer, ako je niz brojeva 1, 2, 3, 4 i bira se 4 broja, onda program $\rightarrow, +, +, \rightarrow, \rightarrow, +, +$ označava kombinaciju $2, 2, 4, 4$. Pitanje, dakle, svodimo na to kako rasporediti $n-1$ strelica i k pluseva, što odgovara pitanju kako rasporediti k pluseva na $n+k-1$ pozicija, tako da je je ukupan broj kombinacija sa ponavljanjem jednak $\binom{n+k-1}{k}$.

Broj particija

Problem: Particija pozitivnog prirodnog broja n je predstavljanje broja n na zbir nekoliko pozitivnih prirodnih brojeva pri čemu je redosled sabiraka nebitan. Na primer particije broja 4 su $1+1+1+1$, $2+1+1$, $2+2$, $3+1$, 4. Napisati program koji određuje broj particija za dati prirodan broj n.

Algoritmi za određivanje broja particija odgovaraju algoritmima za generisanju svih particija.

Svaka particija ima svoj prvi sabirak. Svakoj particiji broja n kojoj je prvi sabirak s (pri čemu je $1 \leq s \leq n$) jednoznačno odgovara neka particija broja $n-s$. Pošto je sabiranje komutativno, da ne bismo suštinski iste particije brojali više puta nametnućemo uslov da sabirci u svakoj particiji budu sortirani (na primer, nerastuće). Zato, ako je prvi sabirak s, svi sabirci iza njega moraju da budu manji ili jednaki od s. Zato nam nije dovoljno samo da umemo da prebrojimo sve particije broja $n - s$, već je potrebno da ojačamo induktivnu hipotezu. Označimo sa $p_{n,s}$ broj particija broja n u kojima su svi sabirci manji ili jednak od s.

- Bazu indukcije čini slučaj $n = 0$, jer broj nula ima samo jednu particiju koja ne sadrži sabirke. Dakle, važi da je $p_{0,s} = 1$. Ako je n veće od nula i $s = 0$, tada ne postoji ni jedna particija, jer od sabiraka koji su svi jednaki nuli (jer svi moraju da budu manji ili jednaki s) ne možemo nikako napraviti neki pozitivan broj. Dakle, za $n > 0$ važi da je $p_{n,0} = 0$.

- Induktivni korak možemo ostvariti na više načina. Najjednostavniji je sledeći. Prilikom izračunavanja $p_{n,s}$ možemo razmatrati dva slučaja: da se u zbiru ne javlja sabirak s ili da se u zbiru javlja sabirak s. Ako se u zbiru ne javlja sabirak s, tada je najveći sabirak $s - 1$ i broj takvih particija je $p_{n,s-1}$. Drugi slučaj je moguć samo kada je $n \geq s$ i broj takvih particija je $p_{n-s,s}$. Na osnovu ovoga, dobijamo narednu rekurzivnu definiciju.

```
// broj particija broja n u kojima su svi sabirci >= smax
int brojParticija(int n, int smax) {
   if (n == 0) return 1;
   if (smax == 0) return 0;
   int broj = brojParticija(n, s-1);
   if (n >= s)
      broj += brojParticija(n-s, s);
   return broj;
}
```

Ova funkcija je neefikasna i može se popraviti dinamičkim programiranjem. Krenimo sa memoizacijom. Uvodimo matricu dimenzije $(n + 1) \times (n + 1)$ koju popunjavamo sa vrednostima -1, čime označavamo da rezultat poziva funkcije još nije poznat. Pre nego što krenemo sa izračunavanjem proveravamo da li je u matrici vrednost različita od -1 i ako jeste, vraćamo tu upamćenu vrednost. Pre svake povratne vrednosti funkcije rezultat pamtimo u matricu.

```
int brojParticija(int n, int smax, vector<vector<int>>& memo) {
  if (memo[n][smax] != -1)
    return memo[n][smax];
  if (n == 0) return memo[n][smax] = 1;
  if (smax == 0) return memo[n][smax] = 0;
  int broj = brojParticija(n, smax-1, memo);
  if (n >= smax)
    broj += brojParticija(n-smax, smax, memo);
  return memo[n][smax] = broj;
```

```
}

int brojParticija(int n) {
  vector<vector<int>> memo(n + 1);
  for (int i = 0; i <= n; i++)
    memo[i].resize(n+1, -1);
  return brojParticija(n, n, memo);
}
```

Ubrzanje je drastično. Za $n = 100$ prva funkcija radi oko $3,862$ sekunde, a druga oko $0,005$ sekundi.

Umesto memoizacije možemo upotrebiti i dinamičko programiranje naviše. Prikažimo tabelu vrednosti funkcije za $n = 10$.

```
n\smax   0  1  2  3  4  5  6  7
0        1  1  1  1  1  1  1  1
1        0  1  1  1  1  1  1  1
2        0  1  2  2  2  2  2  2
3        0  1  2  3  3  3  3  3
4        0  1  3  4  5  5  5  5
5        0  1  3  5  6  7  7  7
6        0  1  4  7  9 10 11 11
7        0  1  4  8 11 13 14 15
```

Na osnovu baze indukcije znamo da će svi elementi prve vrste biti jednaki 1, a da će u prvoj koloni svi elementi osim početnog biti jednaki 0. Jedan od načina da se matrica popunjavam je postepeno uvećavajući vrednost n, tj. popunjavajući vrstu po vrstu.

Element $p_{n,s}$ zavisi od elemenata $p_{n,s-1}$ i (ako je $n \geq s$) $p_{n-s,s}$ i ako se vrste popunjavaju od gore naniže i sleva nadesno, prilikom njegovog izračunavanja oba elementa od kojih zavisi su već izračunata, što daje korektan algoritam.

```
int brojParticija(int N) {
  // alociramo matricu
  vector<vector<int>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(N+1);
  // popunjavamo prvu vrstu
  for (int smax = 0; smax <= N; smax++)
    dp[0][smax] = 1;
  // popunjavamo preostale elemente prve kolone
  for (int n = 1; n <= N; n++)
    dp[n][0] = 0;
  // popunjavamo jednu po jednu vrstu
  for (int n = 1; n <= N; n++)
    for (int smax = 1; smax <= N; smax++) {
      dp[n][smax] = dp[n][smax-1];
      if (n >= smax)
```

```
        dp[n][smax] += dp[n-smax][smax];
    }
  return dp[N][N];
}
```

I vremenska i memorijska složenost ovog algoritma je $O(n^2)$ i ovim redosledom popunjavanja matrice to nije moguće popraviti (jer elementi zavise od elemenata koji se javljaju ne samo u prethodnoj, već i u ranijim vrstama, tako da je potrebno da istovremeno čuvamo sve prethodne vrste). Međutim, ako matricu popunjavamo kolonu po kolonu odozgo naniže, možemo dobiti memorijsku složenost $O(n)$. Naime, svaki element zavisi od elementa u istoj vrsti u prethodnoj koloni i elementa u istoj koloni u nekoj od prethodnih vrsta, tako da ako kolone popunjavamo odozgo naniže, možemo čuvati samo dve uzastopne kolone. Zapravo, možemo čuvati i samo jednu kolonu, ako njeno popunjavanje organizujemo tako da se tokom ažuriranja svi elementi pre tekuće vrste odnose na vrednosti tekuće kolone, a od tekuće vrste do kraja odnose na vrednosti prethodne kolone. Primetimo da se u delu gde je $n < smax$, vrednosti između dve susedne kolone ne menjaju. Time dobijamo narednu optimizovanu implementaciju.

```
int brojParticija(int N) {
  vector<int> dp(N+1, 0);
  dp[0] = 1;
  for (int smax = 1; smax <= N; smax++)
    for (int n = smax; n <= N; n++)
      dp[n] += dp[n-smax];
  return dp[N];
}
```

Broj načina dekodiranja

Problem: Tekst koji se sastoji samo od velikih slova engleske abecede je kodiran tako što je svako slovo zamenjeno njegovim rednim brojem u abecedi. Na primer, tekst `BABAC` je kodiran nizom cifara 21213. Međutim, pošto između slova nije pravljen razmak, dekodiranje nije jednoznačno. Na primer, 21213 može predstaviti `BABAC`, ali i `BAUC`, `BABM`, `BLAC`, `BLM`, `UBAC`, `UUC`, `UBM`. Napiši program koji za određuje broj načina na koji je moguće dekodirati uneti niz cifara.

Ako krenemo da razmatramo niz cifara od njegovog početka, možemo da izdvojimo njegovu prvu cifru, da je dekodiramo i zatim da dekodiramo sufiks dobijen njegovim uklanjanjem, ili da izdvojimo prve dve cifre, dekodiramo njih i zatim da dekodiramo sufiks dobijen njihovim uklanjanjem. Prvi način je moguć ako niz nije prazan i ako je prva cifra različita od nule, dok je drugi moguć ako niz ima bar dve cifre, ako je prva cifra različita od nule, a ta dve cifre grade broj između 1 i 26.

```
long long brojNacinaDekodiranja(const string& s) {
  // prazna niska se dekodira na jedan način
```

```
  if (s == "")
    return 1;
  // rezultat - ukupan broj načina dekodiranja
  long long rez = 0;
  // prvu cifru dekodiramo posebno (ako čini broj između 1 i 9)
  if (s[0] != '0')
    rez += brojNacinaDekodiranja(s.substr(1));
  // prve dve cifre dekodiramo zajedno (ako čine broj između 10 i 26)
  if (s.length() >= 2 && s[0] != '0' && 10*(s[0]-'0')+s[1]-'0' <= 26)
    rez += brojNacinaDekodiranja(s.substr(2));
  // vraćamo rezultat
  return rez;
}
```

Umesto izmene podniske, možemo ga ostaviti konstantnim i samo prosleđivati poziciju i na kojoj počinje sufiks koji trenutno razmatramo. Izlaz iz rekurzije je slučaj praznog niza ($i = n$) koji se može dekodirati na jedan jedini način (praznom niskom).

```
// broj načina dekodiranja sufiksa niske s koji počinje na poziciji i
long long brojNacinaDekodiranja(const string& s, int i) {
  // dužina niske
  int n = s.length();
  // prazan sufiks se može dekodirati samo na jedan način
  if (i == n)
    return 1;
  // rezultat - ukupan broj načina dekodiranja
  long long rez = 0;
  // prvu cifru dekodiramo posebno (ako čini broj između 1 i 9)
  if (s[i] != '0')
    rez += brojNacinaDekodiranja(s, i + 1);
  // prve dve cifre dekodiramo zajedno (ako čine broj između 10 i 26)
  if (i+1 < n && s[i] != '0' && 10*(s[i]-'0')+s[i+1]-'0' <= 26)
    rez += brojNacinaDekodiranja(s, i + 2);
  // vraćamo rezultat
  return rez;
}
```

U ovom rekurzivnom rešenju (bez ozbira na način implementacije) postojaće veoma izvesno veliki broj identičnih rekurzivnih poziva (tj. poziva za istu vrednost ulaznog niza). Npr. ako niz počinje sa 11 tada će se njegov sufiks razmatrati i kada se posebno dekodira svaka od jedinica i kada se dekodira par cifara koji gradi broj 11. Ovo dovodi do nepotrebnog uvećanja složenosti, a kao što znamo, rešenje dolazi u obliku dinamičkog programiranja.

Jedan od načina da smanjimo broj rekurzivnih poziva i nepotrebna izračunavanja je memoizacija. U pomoćnom nizu pamtimo do sada izračunate rezultate (na poziciji i pamtimo vrednost rekurzivnog poziva za parametar i). Niz inicijalizujemo na sve vrednosti -1. Na početku rekurzivne funkcije proveravamo da li

je vrednost na poziciji i različita od -1 i ako jeste, vraćamo je. U suprotnom nastavljamo izvršavanje rekurzivne funkcije. Pre nego što funkcija vrati rezultat upisujemo ga u niz na mesto i.

```
long long brojNacinaDekodiranja(const string& s, int i,
                                vector<long long>& memo) {
  // ako smo ovaj sufiks već obrađivali, vraćamo upamćeni rezultat
  if (memo[i] != -1)
    return memo[i];

  // dužina niske
  int n = s.length();
  // prazan sufiks se može dekodirati samo na jedan način
  if (i == n)
    return memo[n] = 1;
  // rezultat - ukupan broj načina dekodiranja
  long long rez = 0;
  // prvu cifru dekodiramo posebno (ako čini broj između 1 i 9)
  if (s[i] != '0')
    rez += brojNacinaDekodiranja(s, i + 1, memo);
  // prve dve cifre dekodiramo zajedno (ako čine broj između 10 i 26)
  if (i+1 < n && s[i] != '0' && 10*(s[i]-'0')+s[i+1]-'0' <= 26)
    rez += brojNacinaDekodiranja(s, i + 2, memo);
  // pre nego što vratimo rezultat, pamtimo ga
  return memo[i] = rez;
}

long long brojNacinaDekodiranja(const string& s) {
  // alociramo niz dužine n+1 i popunjavamo ga vrednostima -1
  vector<long long> memo(s.length() + 1, -1);
  // dekodiramo celu nisku (sufiks od pozicije 0)
  return brojNacinaDekodiranja(s, 0, memo);
}
```

Možemo se osloboditi rekurzije i niz popuniti odozda naviše.

Neka dp_i predstavlja broj načina da se dekodira sufiks reči s koji počinje na poziciji i (uključujući i nju). S obzirom na strukturu rekurzije, niz dp_i možemo popuniti sdesna nalevo (zaista, pre element na poziciji i zavisi od elemenata na pozicijama $i+1$ i $i+2$).

dp_n predstavlja broj načina da se dekodira prazan sufiks, a to je 1.

U suprotnom u nizu imamo bar jednu cifru.

Ako je s_i cifra `'0'`, tada je $dp_i = 0$ (tada ni prvi, ni drugi način dekodiranja nisu mogući).

Ako je s_i nije cifra `'0'`, treba da razmotrimo mogućnost dekodiranja na prvi i na drugi način. Prvi način je uvek primenljiv (jer sufiks počinje bar jednom cifrom s_i za koju smo utvrdili da nije cifra `'0'`) i u tom slučaju niz možemo

dekodirati na dp_{i+1} načina. Drugi način je primenljiv ako imamo bar dve cifre (ako je $i < n - 1$) i ako je broj dobijen od cifara $s_i s_{i+1}$ (za s_i smo utvrdili da nije cifra `'0'`) između 1 i 26 i u tom slučaju je broj načina dobijen na ovaj način jednak dp_{i+2}. Jednostavnosti radi dp_{n-1} je moguće odrediti posebno, da se ne bi stalno proveravalo da li je $i < n - 1$.

Dakle, $dp_n = 1$, $dp_{n-1} = 0$ ako je s_{n-1} cifra `'0'`, $dp_{n-1} = 1$ ako s_{n-1} nije cifra `'0'`. Za sve vrednosti i od $n - 2$ unatrag važi da je $dp_i = 0$ ako je s_i cifra `'0'`, da je $dp_i = dp_{i+1}$, ako $s_i s_{i+1}$ daju broj koji nije između 1 i 26, i da je $dp_i = dp_{i+1} + dp_{i+2}$ ako $s_i s_{i+1}$ daju broj koji jeste između 1 i 26.

```
long long brojNacinaDekodiranja(const string& s) {
  // dp[i] predstavlja broj nacina da se dekodira sufiks reci s koji
  // pocinje na poziciji i (ukljucujuci i nju).
  int n = s.length();
  vector<long long> dp(n+1);
  // postoji jedan nacin da se dekodira prazan sufiks
  dp[n] = 1;
  // jednocifreni sufiks se moze dekodirati na jedan nacin ako ta
  // cifra nije '0', a na nula nacina inace
  dp[n-1] = s[n-1] != '0';
  // niz dp rekonstruisemo unatrag
  for (int i = n-2; i >= 0; i--) {
    dp[i] = 0;
    // prvu cifru dekodiramo posebno (ako čini broj između 1 i 9)
    if (s[i] == '0')
       dp[i] += dp[i+1];
    // prve dve cifre dekodiramo zajedno (ako čine broj između 10 i 26)
    if (10 * (s[i]-'0') + (s[i+1]-'0') <= 26)
       dp[i] += dp[i+2];
  }
  // traženi rezultat odgovara sufiksu koji počinje na poziciji 0
  return dp[0];
}
```

Primetimo da je za određivanje svake prethodne vrednosti niza dp potrebno znati samo njene dve naredne vrednosti. Stoga nije neophodno čuvati ceo niz, već je u svakom trenutku potrebno poznavati samo dve njegove uzastopne vrednosti (krenuvši od poslednje i pretposlednje).

```
long long brojNacinaDekodiranja(const string& s) {
  int n = s.length();
  // krećemo od dp[n] = 1
  long long dp2 = 1;
  // i od dp[n-1] što je 1 ako s[n-1] nije karakter '0' i 0 ako jeste
  long long dp1 = s[n-1] != '0';
  // niz dp rekonstruišemo unatrag
  for (int i = n-2; i >= 0; i--) {
    // vrednost dp[i]
    long long dp = 0;
```

```
    // prvu cifru dekodiramo posebno (ako čini broj između 1 i 9)
    if (s[i] == '0')
       dp += dp1;
    // prve dve cifre dekodiramo zajedno (ako čine broj između 10 i 26)
    if (10 * (s[i]-'0') + (s[i+1]-'0') <= 26)
       dp += dp2;
    // ažuriramo dva poslednja poznata člana niza
    dp2 = dp1;
    dp1 = dp;
  }
  return dp1;
}
```

Broj sečenja n-tougla na trouglove

Zadatak: Odredi broj načina na koji se n-tougao može dijagonalama koje se ne presecaju raseći na trouglove.

Ovaj broj je određen Katalanovim brojevima zadatim rekurentnim jednačinama $C_0 = 1$, $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$ i važi da se n-toguao može raseći na C_{n-2} načina. Na primer, ako je dat šestougao, tada je broj sečenja jednak $C_4 = C_0C_3 + C_1C_2 + C_2C_1 + C_3C_0$. Fiksirajmo jednu stranicu šestougla. Ona mora biti sigurno stranica nekog trougla u trijangulaciji. Taj trougao može biti odabran tako da mu je teme bilo koje od preostala 4 temena. Ako se uzme prvo, sa jedne njegove strane ne ostaje ništa, a sa druge jedan petougao koji se dalje rekurzivno deli (broj načina da se on podeli je C_3). Ako se uzme drugo teme, sa jedne njegove strane imamo trougao (broj načina da se on podeli je $C_1 = 1$), a sa druge četvorougao (broj načina da se on podeli je $C_2 = 2$) koji se dalje rekurzivno dele. Ako se uzme treće teme, sa jedne njegove strane imamo četvorougao (broj načina da se on podeli je $C_2 = 2$), a sa druge trougao (broj načina da se on podeli je $C_1 = 1$) koji se dalje rekurzivno dele. Na kraju, ako se uzme četvrto teme, sa jedne njegove strane ostaje petougao (broj načina da se on podeli je C_3), a sa druge ništa. Rekurentna formula sledi lako na osnovu ovog razmatranja. Veoma slično se pokazuje i za opšti slučaj.

Na osnovu ovoga je jednostavno dati rekurzivnu formulaciju.

```
int katalan(int n) {
   if (n == 0) return 1;
   int zbir = 0;
   for (int i = 0; i <= n; i++)
      zbir += katalan(i) * katalan(n-i);
   return zbir;
}
```

I ovde dolazi do preklapanja rekurzivnih poziva, pa je stoga bolje napraviti rešenje dinamičkim programiranjem. Prikažimo odmah rešenje naviše.

```
int katalan(int N) {
   vector<int> dp(N);
   dp[0] = 1;
   for (int n = 1; n <= N; n++) {
      for (int i = 0; i <= n; i++)
         zbir += dp[i] + dp[n-i];
      dp[n] = zbir;
   }
   return dp[N];
}
```

Pošto u ovoj rekurentnoj vezi svaki Katalanov broj zavisi od svih prethodnih, nije moguće napraviti memorijsku optimizaciju, pa je memorijska složenost $O(n)$, a vremenska $O(n^2)$.

Moguće je dokazati da važi da je $C_n = \frac{1}{n+1}\binom{2n}{n}$. U tom slučaju se Katalanov broj C_n može izračunati u vremenskoj složenosti $O(n^2)$ i memorijskoj $O(1)$.

Zbir podskupa (varijanta 0-1 ranca)

Umesto izračunavanja broja kombinatornih objekata, ponekad se možemo pitati da li postoji i jedan objekat koji zadovoljava neko dato svojstvo. Ilustrujmo ovo narednim primerom.

Problem: Dato je n predmeta čije su mase $m_0, \dots m_{n-1}$ i ranac nosivosti M (svi ti brojevi su prirodni). Napisati program koji određuje da li se ranac može ispuniti do kraja nekim od n datih predmeta (tako da je zbir masa predmeta jednak nosivosti ranca).

Rešenje grubom silom bi podrazumevalo da se isprobaju svi podskupovi predmeta. Složenost tog pristupa bi bila $O(2^n)$. Naglasimo da smo ovaj prisup već implementirali u delu pretrazi, no sada ćemo ga unaprediti, pod pretpostavkom da su nosivost ranca i mase predmeta relativno mali prirodni brojevi.

Već smo videli da se može organizovati pretraga sa povratkom, čiji su parametri dužina niza predmeta i preostala nosivost ranca. U svakom razmatramo mogućnost u kojoj poslednji predmet nije stavljen u ranac i mogućnost u kojoj poslednji predmet jeste stavljen u ranac (pod pretpostavkom da može da stane).

```
bool zbirPodskupa(vector<int>& m, int n, int M) {
   // preostala nosivost je 0, pa je ranac kompletno popunjen
   if (M == 0)
      return true;
   // preostala nosivost je pozitivna, ali nemamo više predmeta
   // pa je ne možemo popuniti
   if (n == 0)
      return false;
   // preskačemo n-ti predmet i pokušavamo da ranac napunimo sa
   // prvih n-1 predmeta
```

```
    if (zbirPodskupa(m, n-1, M))
       return true;
    // ako n-ti predmet može da stane u ranac stavljamo ga i
    // pokušavamo da preostalu nosivost popunimo pomoću preostalih
    // n-1 predmeta
    if (m[n-1] <= M && zbirPodskupa(m, n-1, M - m[n-1]))
       return true;
    // ako ne možemo da napravimo podskup ni sa ni bez n-tog predmeta
    // onda podskup nije moguće napraviti
    return false;
}


bool zbirPodskupa(vector<int>& m, int M) {
    return zbirPodskupa(m, m.size(), M);
}
```

Implementacija je i jednostavnija a složenost najgoreg slučaja je eksponencijalna. Isto bi ostalo i ako bismo primenili i ostala sečenja koja smo ilustrovali u delu o pretrazi.

Veliki problem ovog pristupa je to što se isti rekurzivni pozivi ponavljaju više puta (pre svega zahvaljući činjenici da su nosivost ranca i mase predmeta celobrojni). Na primer, ako su poslednja tri od 10 predmeta mase 3, 5 i 2, ako je polazna nosivost ranca 15 i ako broj predmeta i preostalu nosivost ranca u rekurzivnom pozivu označimo uređenim parom (n, M), tada se dolazi do sledećeg drveta rekurzivnih poziva.

```
                                  (10, 15)
              (9, 15)                                     (9, 13)
     (8, 15)            (8, 10)           (8, 13)             (8, 8)
(7, 15) (7, 12)  (7, 10) (7, 7)   (7, 13) (7, 10)  (7, 8) (7, 5)
```

Primećujemo da se poziv za par argumenata $(7, 10)$ ponavlja i u levom i u desnom delu drveta pretrage. U levom delu to je posledica uzimanja predmeta mase 5 i izostavljanja predmeta mase 2 i 3, dok je u desnom delu to posledica uzimanja predmeta mase 2 i 3 i izostavljanja predmeta mase 5. Pošto je u slučaju celobrojnih masa prilično verovatno da će postojati različiti podskupovi predmeta iste zbirne mase, vrlo je verovatno da će dosta rekurzivnih poziva biti ponovljeno. Ubrzanje se stoga vrlo verovatno može dobiti dinamičkim programiranjem.

Ako želimo da memorišemo rezultate poziva funkcije za razne parametre, vidimo da su parametri koji se menjaju tokom rekurzije `n` i `M`. Ako želimo da možemo da pamtimo sve njihove kombinacije možemo upotrebiti matricu koja je dimenzije $(n + 1) \times (M + 1)$, gde je n početni broj predmeta, a M je nosivost celog ranca. Pošto nismo sigurni da će cela matrica biti popunjena, moguće je memoizaciju organizovati tako da se podaci čuvaju u mapi koja preslikava parove (n, M) u rezultujuće logičke vrednosti. Time se može uštedeti memorija, ali je implementacija za prilično značajni konstantni faktor sporija nego kada se koristi

matrica.

```
bool zbirPodskupa(vector<int>& m, int n, int M,
                  unordered_map<pair<int, int>, bool, pairhash>& memo) {
  auto p = make_pair(n, M);
  auto it = memo.find(p);
  if (it != memo.end())
    return it->second;

  if (M == 0) return true;
  if (n == 0) return false;
  if (zbirPodskupa(m, n-1, M, memo))
    return memo[p] = true;
  if (m[n-1] <= M && zbirPodskupa(m, n-1, M - m[n-1], memo))
    return memo[p] = true;
  return memo[p] = false;
}

bool zbirPodskupa(vector<int>& m, int M) {
   map<pair<int, int>, bool> memo;
   return zbirPodskupa(m, m.size(), M, memo);
}
```

Program možemo implementirati i dinamičkim programiranjem naviše. Matricu možemo popunjavati u rastućem redosledu broja predmeta tj. možemo obrađivati jedan po jedan predmet u redosledu u kom su dati.

```
bool zbirPodskupa(const vector<int>& masa, int M) {
  int N = masa.size();
  vector<vector<int>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(M + 1);

  dp[0][0] = true;
  for (int m = 1; m <= M; m++)
    dp[0][m] = false;

  for (int n = 1; n <= N; n++) {
    dp[n][0] = true;
    for (int m = 0; m <= M; m++)
      dp[n][m] = dp[n-1][m] ||
                 masa[n-1] <= m && dp[n-1][m - masa[n-1]];
  }

  return dp[N][M];
}
```

Za niz predmeta $3, 1, 7, 2$ i masu 6 dobija se sledeća matrica (vrednost $\top$ je označena sa 1, a $\bot$ sa 0).

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

Naravno, pretragu možemo zaustaviti čim vrednost u koloni M prvi put postane tačna (jer će se tada i u svim narednim vrstama u toj koloni nalaziti vrednost tačno).

Na osnovu popunjene matrice možemo jednostavno rekonstruisati rešenje, tj. odrediti neki podskup čiji je zbir jednak datom broju. Krećemo iz donjeg desnog ugla. Ako se u njemu ne nalazi vrednost tačno (jedinica), podskup ne postoji. U suprotnom tražimo prvu vrstu u kojoj se ta jedinica javila. Taj predmet mora biti uključen u podskup i nakon toga nastavljamo da na isti način određujemo podskup čiji je zbir jednak zbiru bez tog predmeta.

U tekućem primeru prva vrsta u kojoj se pojavljuje jedinica u poslednjoj koloni (kojoj odgovara masa 6) je poslednja vrsta, pa je neophodno uključiti poslednji predmet sa masom 2. Nakon toga prelazimo na kolonu kojoj odgovara masa 4 i prva vrsta u kojoj se tu javlja jedinica je vrsta broj 2 kojoj odgovara predmet mase 1. Njega uključujemo u podskup i nastavljamo analizu od kolone kojoj odgovara masa 3. Prva vrsta u kojoj se u toj koloni javlja jedinica je vrsta broj 1 kojoj odgovara predmet sa masom 3. I taj predmet uključujemo u podskup i pošto bi se nakon toga nastavilo sa kolonom kojoj odgovara masa 0, pronašli smo traženi podskup.

Implementacija postupka rekonstrukcije rešenja može biti sledeća.

```
vector<int> nadjiPodskup(const vector<int>& masa,
                         const vector<vector<int>>& dp, int n, int m) {
  vector<int> podskup;
  while (m > 0) {
    while (dp[n][m])
      n--;
    podskup.push_back(masa[n]);
    m -= masa[n];
  }
  return podskup;
}
```

Kada nam nije bitno da rekonstruišemo rešenje, tada možemo napraviti memorijsku optimizaciju. Kada se matrica popunjava vrstu po vrstu, elementi svake naredne vrste matrice zavise samo od elemenata prethodne vrste i to onih koji se nalaze levo od njih. Zato možemo održavati samo jednu tekuću vrstu i možemo je ažurirati zdesna nalevo.

```
bool zbirPodskupa(vector<int>& masa, int M) {
  int N = masa.size();
  vector<bool> dp(M+1, false);

  dp[0] = true;
  for (int n = 1; n <= N; n++)
    for (int m = M; m >= masa[n-1]; m--)
      dp[m] = dp[m] || dp[m - masa[n-1]];

  return dp[M];
}
```

Razmislimo šta smo ovim transformacijama zapravo dobili. Tekuća vrsta u svakom koraku kodira skup masa ranaca koje je moguće dobiti od predmeta zaključno sa tekućim. Razmatramo svaku moguću masu i zaključujemo da je možemo dobiti ili tako što smo je već ranije mogli dobiti ili tako što smo na neku ranije dobijenu masu mogli dodati masu n-tog predmeta. Na osnovu ovoga možemo dobiti i malo drugačiju implementaciju.

```
bool zbirPodskupa(vector<int>& masa, int M) {
  int N = masa.size();
  vector<bool> dp(M+1, false);

  dp[0] = true;
  for (int n = 1; n <= N; n++)
    for (int m = M - masa[n-1]; m >= 0; m--)
      if (dp[m])
         dp[m + masa[n-1]] = true;

  return dp[M];
}
```

Ako bismo skup umesto vektora bitova predstavili pomoću objekta tipa `set`, dobili bismo dosta sporiju implementaciju.

```
bool zbirPodskupa(vector<int>& masa, int M) {
  int N = masa.size();
  set<int> s;

  s.insert(0);
  for (int n = 1; n <= N; n++)
    for (auto it = s.rbegin(); it != s.rend(); it++)
      if (*it + masa[n-1] <= M)
        s.insert(*it + masa[n-1]);

  return s.find(M) != s.end();
}
```

Analizirajmo složenost algoritma dinamičkog programiranja (kada se koristi vektor logičkih vrednosti koji se interno najčešće predstavlja nizom bitova).

Memorijska složenost je $O(M)$, dok je vremenska složenost $O(n \cdot M)$. Na prvi pogled se može pomisliti da je u pitanju polinomijalna složenost, međutim, to ne bi bilo tačno. Naime, složenost zavisi od *vrednosti* parametra M. Ako je M 32-bitni neoznačen ceo broj ta vrednost može biti $2^{32} - 1$ (oko 4 milijarde), što je već nedopustivo veliko i za savremene računare. Ako bismo upotrebili 64-bitni neoznačen ceo broj ta vrednost može biti $2^{64} - 1$. Veličina ulaza je određena brojem bitova potrebnih za zapis broja M, a maksimalna vrednost broja očigledno eksponecijalno zavisi od te veličine. Dakle, u pitanju je i dalje algoritam čija vremenska (a i memorijska) složenost eksponencijalno zavisi od veličine ulaza. Ipak, za relativno male vrednosti M, ovaj algoritam se ponaša efikasno. Ovakvi algoritmi se nazivaju *pseudo-polinomijalni.* Složenost postupka rekonstrukcije je $O(M + n)$.

Rešavanje optimizacionih problema dinamičkim programiranjem

Tehnika dinamičkog programiranja se često koristi i za rešavanje optimizacionih problema. Ponovo je ključan korak induktivno-rekurzivna konstrukcija, dok dinamičko programiranje služi da se reši problem nepotrebnih višestrukih izračunavanja istih vrednosti (tzv. preklapajućih rekurzivnih poziva).

Kod rešavanja optimizacionih problema induktivno-rekurzivnom konstrukcijom jako je važno uveriti se da problem ima tzv. *svojstvo optimalne podstrukture* (engl. optimal substrucure property), koje nam garantuje da će se optimalno rešenje problema nalaziti na osnovu optimalnih rešenja potproblema.

Na primer, ako određujemo najkraći put između gradova i ako najkraći put od Beograda do Subotice prelazi preko Novog Sada, onda on mora da uključi najkraći put od Novog Sada do Subotice. Naime, ako bi postojao kraći put od Novog Sada do Subotice, onda bi se deonica od Novog Sada do Subotice u optimalnom putu od Beograda do Subotice preko Novog Sada mogla skratiti, što je u kontradikciji sa pretpostavkom da je put od Beograda do Subotice optimalan.

Nemaju svi problemi svojstvo optimalne podstrukture. Na primer, ako je najjeftiniji let od Beograda do Njujorka preko Minhena i Pariza, ne možemo da zaključimo da će najjeftiniji put od Beograda do Pariza biti preko Minhena, jer avio kompanije daju popuste na različite kombinacije letova.

Minimalni broj novčića

Krenimo od narednog jednostavnog uopštenja problema ranca tj. zbira podskupa koji smo prethodno razmotrili.

Problem: Na raspolaganju imamo n novčića čiji su iznosi celi brojevi $m_0, \ldots, m_{n-1}$. Napisati program koji određuje minimalni broj novčića pomoću

kojih se može platiti dati iznos (svaki novčić se može upotrebiti samo jednom).

Ponovo su moguća rešenja grubom silom (provera svih podskupova novčića) i pretragom sa odsecanjima (pri čemu je osnovno odsecanje ono u kome se pretraga prekida kada je iznos koji treba formirati manji od trenutnog zbira uzetih novčića ili je veći od zbira preostalih novčića). Jednostavnosti implementacije radi, ako plaćanje nije moguće pretpostavićemom da je broj potrebnih novčića $+\infty$.

U srcu algoritma je induktivno-rekurzivna konstrukcija po broju novčića u nizu koji razmatramo. Razmatramo opciju u kojoj poslednji novčić u tom nizu ne učestvuje i opciju u kojoj poslednji novčić učestvuje u optimalnom plaćanju. U prvom slučaju potrebno je odrediti minimalni broj novčića iz prefiksa niza bez poslednjeg elementa kojima se može platiti polazni iznos (jasno je da je potrebno naći rešenje sa najmanjim brojem novčića, ako je to moguće, te se u ovom slučaju traži optimalno podrešenje). Ako je njihov zbir manji od iznosa, pretraga se može iseći, jer plaćanje nije moguće. Poslednji novčić može biti deo nekog plaćanja datog iznosa samo ako je njegova vrednost manja ili jednaka od poslednjeg iznosa. U tom slučaju potrebno je pomoću ostalih novčića iz niza platiti iznos umanjen za vrednost poslednjeg novčića i to je neophodno uraditi sa najmanjim brojem novčića (i ovde se traži optimalno podrešenje). Rešenje koje bi uključilo poslednji novčić, a u kome bi se ostatak formirao pomoću više novčića nego što je potrebno, ne bi moglo da bude optimalno, jer bi se plaćanje ostatka moglo zameniti sa manjim brojem novčića i uključivanjem poslednjeg novčića bi se dobilo bolje rešenje polaznog problema. Dakle, u oba slučaja ovaj problem zadovoljava uslov optimalne podstrukture. Tražimo minimalni broj novčića M_{bez} da se plati ceo iznos pomoću novčića bez poslednjeg. Ako je vrednost novčića veća od iznosa tada je M_{bez} konačno rešenje, a u suprotnom određujemo minimalni broj novčića potreban da se plati iznos umanjen za vrednost poslednjeg novčića i njegovim uvećavanjem za 1 dobijamo minimalni broj novčića M_{sa} potreban da se plati iznos kada je poslednji novčić uključen. Konačno rešenje je manji od brojeva M_{bez} i M_{sa}.

Ako sa $M(n, I)$ obeležimo minimalni broj novčića potrebnih da se plati iznos I pomoću prvih n novčića iz datog niza, tada dobijamo sledeću rekurzivnu formulaciju.

$$
\begin{aligned}
&M(n,0) = 0,\\
&M(0,I) = +\infty, \quad \text{za } I > 0\\
&M(n,I) = M(n-1,I), \quad \text{za } m_{n-1} > I, I > 0, n > 0\\
&M(n,I) = \min\left(M(n-1,I), 1 + M(n-1, I - m_{n-1})\right), \quad \text{za } m_{n-1} \leq I, I > 0, n > 0
\end{aligned}
$$

U nastavku ćemo prikazati rešenje dinamičkim programiranjem. Krenimo od memoizovane verzije pretrage sa odsecanjem. Za memoizaciju možemo upotrebiti matricu (pošto je i u ovom primeru moguće da matrica bude retko popunjena, moguće je i upotrebiti i heš-mapu).

```
// tip matrice koji ćemo koristiti pri memoizaciji
typedef vector<vector<int>> Memo;

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

int najmanjiBrojNovcica(const vector<int>& novcici, int n,
                        int iznos, int preostalo, Memo& memo) {
  if (iznos == 0)
    return 0;
  if (n == 0)
    return INF;

  if (memo[n][iznos] != -1)
    return memo[n][iznos];

  if (preostalo < iznos)
    return memo[n][iznos] = INF;

  int rez = najmanjiBrojNovcica(novcici, n-1, iznos,
                                preostalo - novcici[n-1], memo);
  if (novcici[n-1] <= iznos) {
    int pom = najmanjiBrojNovcica(novcici, n-1, iznos - novcici[n-1],
                                  preostalo - novcici[n-1], memo);
    if (pom != INF)
      rez = min(rez, 1 + pom);
  }

  return memo[n][iznos] = rez;
}

int najmanjiBrojNovcica(const vector<int>& novcici, int iznos) {
  Memo memo(novcici.size() + 1);
  for (int i = 0; i <= novcici.size(); i++)
    memo[i].resize(iznos + 1, -1);

  int zbir = 0;
  for (int i = 0; i < novcici.size(); i++)
    zbir += novcici[i];
  return najmanjiBrojNovcica(novcici, novcici.size(), iznos, zbir, memo);
}
```

Naravno, moguće je i rešenje odozdo naviše.

```
int najmanjiBrojNovcica(const vector<int>& novcici, int iznos) {
  int N = novcici.size(), M = iznos;
  vector<vector<int>> dp(N + 1);
  for (int i = 0; i <= novcici.size(); i++)
    dp[i].resize(M + 1);

  dp[0][0] = 0;
```

```
  for (int iznos = 1; iznos <= M; iznos++)
    dp[0][iznos] = INF;
  for (int n = 1; n <= N; n++) {
    dp[n][0] = 0;
    for (int iznos = 1; iznos <= M; iznos++) {
      dp[n][iznos] = dp[n-1][iznos];
      if (novcici[n-1] <= iznos && dp[n-1][iznos-novcici[n-1]] != INF)
        dp[n][iznos] = min(dp[n][iznos], 1 + dp[n-1][iznos-novcici[n-1]]);
    }
  }
  return dp[N][M];
}
```

Matrica za primer niza novčića $3, 7, 2, 4, 6$ i iznosa od 15 dinara je sledeća.

```
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
    0   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
3   0|  .   .   1   .   .   .   .   .   .   .   .   .   .   .   .
7   0   .   .   1   .   .   .   1|  .   .   2   .   .   .   .   .
2   0   .   1   1   .   2   .   1   .   2|  2   .   .   3   .   .
4   0   .   1   1   1   2   2   1   .   2|  2   2   .   3   3   .
6   0   .   1   1   1   2   1   1   2   2   2   2   3   2   3   3|
```

Kada je izračunata matrica, rekonstrukciju rešenja možemo izvršiti veoma jednostavno. Krećemo iz donjeg desnog ugla i u svakom trenutku proveravamo da li je broj jednak onom iznad sebe ili onom elementu prethodne vrste koji se dobija umanjivanjem tekućeg iznosa za vrednost tekućeg novčića (moguće je i da je jednak obema tim vrednostima i tada su moguća različita rešenja, pa se možemo opredeliti za bilo koje od njih). U prethodnom primeru su označene vrednosti koje se koriste tokom rekonstrukcije. Iznos 15 se gradi od 3 novčića iz celog skupa. Pošto se iznad broja 3 ne nalazi trojka moramo uzeti novčić 6 i prelazimo na rekonstrukciju iznosa 9 pomoću 2 novčića iz skupa koji sadrži prva 4 novčića. Iznos 9 je moguće rekonstruisati pomoću 2 novčića iz skupa koji sadrži prva 3 novčića, a iznos 5 nije moguće rekonstruisati pomoću jednog novčića iz tog skupa, što znači da novčić 4 moramo preskočiti. Pošto iznos 9 nije moguće rekonstruisati samo pomoću prva dva novčića, nović 2 moramo uzeti i nakon toga rekonstruišemo iznos 7 pomoću prva dva novčića. Pošto 7 nije moguće formirati samo od prvog novčića, moramo uzeti novčić 7, nakon čega stižemo do iznosa 0. Time se rekonstrukcija završava i odabrali smo novčiće $7 + 2 + 6 = 15$.

```
vector<int> resenje(const vector<int>& novcici, int n, int iznos,
                    const vector<vector<int>>& dp) {
  vector<int> resenje;
  while (iznos > 0) {
    if (dp[n - 1][iznos] == dp[n][iznos])
      n--;
    else {
      resenje.push_back(novcici[n-1]);
      iznos -= novcici[n-1];
```

```
      n--;
    }
  }
}
```

Veoma jednostavno možemo i nabrojati sva rešenja. Rešenja popunjavamo u vektor `resenje`, pri čemu je `k` broj njegovih trenutno popunjenih elemenata.

```
void ispisiSvaResenja(const vector<int>& novcici, int n, int iznos,
                      const vector<vector<int>>& dp,
                      vector<int>& resenje, int k) {
  if (iznos == 0)
    ispis(resenje);
  else {
    // da li postoji optimalno rešenje bez n-tog novčića
    if (dp[n-1][iznos] == dp[n][iznos])
      ispisiSvaResenja(novcici, n-1, iznos, dp, resenje, k);
    // da li postoji optimalno rešenje sa n-tim novčićem
    if (iznos >= novcici[n-1] &&
        dp[n][iznos] == 1 + dp[n-1][iznos - novcici[n-1]]) {
      resenje[k] = novcici[n-1];
      ispisiSvaResenja(novcici, n-1, iznos - novcici[n-1], dp,
                       resenje, k + 1);
    }
  }
}

void ispisiSvaResenja(const vector<int>& novcici, int n, int iznos,
                      const Memo& dp) {
  // vektor u koji se skladišti tekuće rešenje
  vector<int> resenje(dp[n][iznos]);
  // pokrećemo pretragu
  ispisiSvaResenja(novcici, n, iznos, dp, resenje, 0);
}
```

Još jedan interesantan zadatak može biti izračunavanje broja različitih najkraćih rešenja. Prethodna funkcija se veoma jednostavno modifikuje tako da ne ispisuje, već da broji rešenja (u pitanju je prebrojavanje kombinatornih objekata, o kome je već bilo reči). Naravno, prilikom prebrojavanja će doći do ponavljanja rekurzivnih poziva, pa je potrebno da ponovo primenimo dinamičko programiranje. Najjednostavnije je primeniti memoizaciju (a dinamičko programiranje naviše vam ostavljamo za vežbu). Primetimo da se u tom slučaju u rešenju zadatka dva puta primenjuje dinamičko programiranje. Broj rešenja jako brzo raste, pa treba biti obazriv oko prekoračenja.

```
long long brojResenja(const vector<int>& novcici, int n, int iznos,
                      vector<vector<int>>& dp, Memo& memo) {
  if (iznos == 0)
    return 1;
```

```
  else {
    if (memo[n][iznos] != -1)
      return memo[n][iznos];

    long long broj = 0;
    // broj rešenja u kojima ne učestvuje n-ti novčić
    if (dp[n-1][iznos] == dp[n][iznos])
      broj += brojResenja(novcici, n-1, iznos, dp, memo);
    // broj rešenja u kojima učestvuje n-ti novčić
    if (iznos >= novcici[n-1] &&
        dp[n][iznos] == 1 + dp[n-1][iznos - novcici[n-1]])
      broj += brojResenja(novcici, n-1, iznos - novcici[n-1], dp, memo);
    return memo[n][iznos] = broj;
  }
}

long long brojResenja(const vector<int>& novcici, int n, int iznos,
                      vector<vector<int>>& dp) {
  // alociramo matricu za memoizaciju broja rešenja
  Memo memo(n+1);
  for (int i = 0; i <= n; i++)
    memo[i].resize(iznos + 1, -1);
  return brojResenja(novcici, n, iznos, dp, memo);
}
```

Ako nas ne zanima rekonstrukcija rešenja, tada možemo izvršiti memorijsku implementaciju i možemo pamtiti samo tekuću vrstu matrice. I u ovoj implementaciji možemo napraviti odsecanje u slučaju kada je zbir svih novčića manji od iznosa koji treba naplatiti.

```
int najmanjiBrojNovcica(const vector<int>& novcici, int iznos) {
  int N = novcici.size(), M = iznos;
  vector<int> dp(M + 1);

  int zbir = 0;
  dp[0] = 0;
  for (int iznos = 1; iznos <= M; iznos++)
    dp[iznos] = INF;
  for (int n = 1; n <= N; n++) {
    zbir += novcici[n-1];
      for (int iznos = min(M, zbir); iznos >= novcici[n-1]; iznos--) {
      if (dp[iznos-novcici[n-1]] != INF)
        dp[iznos] = min(dp[iznos], 1 + dp[iznos-novcici[n-1]]);
    }
  }
  return dp[M];
}
```

Maksimalni zbir segmenta

Razmotrimo problem određivanja maksimalnog zbira segmenta iz ugla dinamičkog programiranja.

Problem: Definisati efikasnu funkciju koja pronalazi najveći mogući zbir segmenta (podniza uzastopnih elemenata) datog niza brojeva. Proceni joj složenost.

Pristupimo problemu induktivno-rekurzivno. Za svaku poziciju $0 \leq i \leq n$ odredimo vrednost najvećeg zbira segmenta niza određenog pozicijama iz intervala oblika $[j, i)$ za $0 \leq j \leq i$, tj. najveću vrednost sufiksa koji se završava neposredno pre pozicije i.

- Bazni slučaj je $i = 0$ i tada je $j = 0$ jedini mogući izbor za j, što odgovara praznom sufiksu čiji je zbir 0.
- Pretpostavimo da želimo da odredimo ovu vrednost za neko $0 < i \leq n$. Vrednost j može biti ili jednaka i ili neka vrednost strogo manja od i. Ukoliko je $j = i$, tada je u pitanju prazan sufiks čiji je zbir nula. U suprotnom se zbir sufiksa može razložiti na zbir elemenata na pozicijama $[j, i-1)$ i na element a_{i-1}. Zbir a_{i-1} je fiksiran, pa da bi ovaj zbir bio maksimalni, potrebno je da zbir elemenata na pozicijama $[j, i-1)$ bude maksimalni, međutim, on je sufiks pre pozicije $i-1$, pa maksimalnu vrednost tog zbira znamo na osnovu induktivne hipoteze. Maksimalni zbir je dakle veći broj između tog zbira i zbira praznog segmenta, tj. nule.

Time dobijamo narednu rekurzivnu definiciju u kojoj dolazi do preklapanja rekurzivnih poziva.

```
int maksimalniZbirSegmenta(const vector<int>& a, int i) {
   if (i == 0)
      return -1;
   return max(0, maksimalniZbirSegmenta(a, i-1) + a[i-1]);
}
```

Funkcija sama po sebi nije puno korisna, jer nas zanima maksimalna vrednost segmenta, a ne maksimalna vrednost sufiksa. Međutim pošto se svaki segment javlja u nekom trenutku kao sufiks, možemo ojačati induktivnu hipotezu i funkciju prilagoditi tako da uz maksimum sufiksa vraća i maksimum svih segmenata pre te pozicije. Međutim, ako upotrebimo dinamičko programiranje naviše, za tim nema potrebe, jer nakon popunjavanja niza maksimuma sufiksa, možemo njegov maksimum lako odrediti u jednom dodatnom prolasku.

```
int maksimalniZbirSegmenta(const vector<int>& a) {
   int n = a.size();
   // za svaku poziciju određujemo maksimalni zbir sufiksa
   // koji se završava na toj poziciji
   vector<int> dp(n + 1);
   dp[0] = 0;
   for (int i = 1; i <= n; i++)
```

```
      dp[i] = max(0, dp[i-1] + a[i-1]);
   // određujemo maksimalni zbir segmenta (on je sigurno zbir
   // nekog sufiksa, pa tražimo maksimum svih zbirova sufiksa)
   int rez = dp[0];
   for (int i = 1; i <= n; i++)
      rez = max(rez, dp[i]);
   return rez;
}
```

Ako je niz `a` jednak `-2 3 2 -3 -3 -2 4 5 -8 3`, tada niz `dp` postaje `0 3 5 2 0 0 4 9 1 4` i maksimum mu je `9`.

Naravno, dva prolaska možemo objediniti u jedan.

```
int maksimalniZbirSegmenta(const vector<int>& a) {
   int n = a.size();
   vector<int> dp(n + 1);
   dp[0] = 0;
   int rez = dp[0];
   for (int i = 1; i <= n; i++) {
      dp[i] = max(0, dp[i-1] + a[i-1]);
      rez = max(rez, dp[i]);
   }
   return rez;
}
```

I vremenska i memorijska složenost ovog algoritma je $O(n)$.

Pošto tekuća vrednost u nizu zavisi samo od prethodne, niz nam zapravo nije potreban i možemo čuvati samo tekuću vrednost u nizu čime memorijsku složenost možemo spustiti na $O(1)$.

```
int maksimalniZbirSegmenta(const vector<int>& a) {
   int n = a.size();
   int dp = 0;
   int rez = dp;
   for (int i = 1; i <= n; i++) {
      dp = max(0, dp + a[i-1]);
      rez = max(rez, dp);
   }
   return rez;
}
```

Promenljive možemo preimenovati u skladu sa njihovom semantikom i tako dobiti Kadanov algoritam koji smo i ranije izveli, bez eksplicitnog pozivanja na tehniku dinamičkog programiranja.

```
int maksimalniZbirSegmenta(const vector<int>& a) {
   int n = a.size();
   int maksSufiks = 0;
```

```
   int maksSegment = maksSufiks;
   for (int i = 1; i <= n; i++) {
      maksSufiks = max(0, maksSufiks + a[i-1]);
      maksSegment = max(maksSegment, maksSufiks);
   }
   return maksSegment;
}
```

Najduži zajednički podniz

Problem: Defisati funkciju koja određuje dužinu najduže zajedničke podniske (ne obavezno uzastopnih karaktera) dve date niske. Na primer za niske `ababc` i `babbca` najduža zajednička podniska je `babc`.

Krećemo od rešenja induktivno-rekurzivnom konstrukcijom.

- Ako je bilo koja od dve niske prazna, tada je jedini njen podniz prazan, pa je dužina najdužeg zajedničkog podniza jednaka nuli.
- Ako su obe niske neprazne, tada možemo uporediti njihova poslednja slova. Ako su ona jednaka, mogu biti uključena u najduži zajednički podniz i problem se rekurzivno svodi na pronalaženje najdužeg zajedničkog podniza njihovih prefiksa. U suprotnom, nije moguće da oba poslednja slova budu uključena u zajednički podniz. Zato razmatramo najduži zajednički podniz prve niske i prefiksa druge niske bez njenog poslednjeg slova i zajednički podniz druge niske i prefiksa prve niske bez njenog poslednjeg slova. Duži od dva podniza biće najduži zajednički podniz te dve niske. Naglasimo i da nije neophodno razmatrati najduži zajednički podniz dva prefiksa, jer se proširivanjem nekog od dva prefiksa za poslednje slovo samo ne može dobiti podniz koji bi bio kraći. Takođe, naglasimo da je u ovom problemu zadovoljeno svojstvo optimalne podstrukture (jer bi se pronalaženjem neoptimalnih rešenja za prefikse dobila neoptimalna rešenja za polazne niske).

Pošto rekurzija teče po prefiksima niski, jedini promenljivi parametri tokom rekurzije mogu biti dužine tih prefiksa. Ako sa $f(m, n)$ označimo dužinu najdužeg zajedničkog podniza prefiksa niske a dužine m i prefiksa niske b dužine n, tada važi sledeća rekurentna veza:

$$
\begin{aligned}
f(0, n) &= 0\\
f(m, 0) &= 0\\
f(m, n) &= f(m-1, n-1) + 1, \quad \text{za } m, n > 0 \text{ i } a_{m-1} = b_{n-1}\\
f(m, n) &= \max\left(f(m, n-1), f(m-1, n)\right), \quad \text{za } m, n > 0 \text{ i } a_{m-1} \neq b_{n-1}
\end{aligned}
$$

```
int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2) {
```

```
    if (n1 == 0 || n2 == 0)
       return 0;
    if (s1[n1-1] == s2[n2-1])
       return najduziZajednickiPodniz(s1, n1-1, s2, n2-1) + 1;
    else
       return max(najduziZajednickiPodniz(s1, n1, s2, n2-1),
                  najduziZajednickiPodniz(s1, n1-1, s2, n2));
    return rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  return najduziZajednickiPodniz(s1, n1, s2, n2);
}
```

U direktnom rekurzivnom rešenju ima mnogo preklapajućih rekurzivnih poziva. Stoga je efikasnost moguće popraviti tehnikom dinamičkog programiranja. Jedan mogući pristup je da upotrebimo memoizaciju. Vrednost dužine najdužeg podniza za svaki par dužina prefiksa možemo pamtiti u matrici.

```
int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2,
                            vector<vector<int>>& memo) {
  if (memo[n1][n2] != -1)
    return memo[n1][n2];

  if (n1 == 0 || n2 == 0)
    return memo[n1][n2] = 0;

  int rez;
  if (s1[n1-1] == s2[n2-1])
    rez = najduziZajednickiPodniz(s1, n1-1, s2, n2-1, memo) + 1;
  else
    rez = max(najduziZajednickiPodniz(s1, n1, s2, n2-1, memo),
              najduziZajednickiPodniz(s1, n1-1, s2, n2, memo));
  return memo[n1][n2] = rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> memo(n1+1);
  for (int i = 0; i <= n1; i++)
    memo[i].resize(n2 + 1, -1);

  return najduziZajednickiPodniz(s1, n1, s2, n2, memo);
}
```

Problem preklpajućih rekurzivnih poziva se može rešiti ako se upotrebi dinamičko programiranje naviše. Dužine najdužih podnizova prefiksa možemo čuvati u matrici. Element matrice na poziciji (m, n) zavisi samo od elemenata na pozicijama

$(m-1, n)$, $(m, n-1)$ i $(m-1, n-1)$, tako da matricu požemo da popunjavamo bilo vrstu po vrstu, bilo kolonu po kolonu. Prikažimo matricu za primer niske `xmjyauz` i `mzjawxu`.

```
      m z j a w x u
    0 1 2 3 4 5 6 7
   ----------------
  0|0 0 0 0 0 0 0 0
x 1|0 0 0 0 0 0 1 1
m 2|0 1 1 1 1 1 1 1
j 3|0 1 1 2 2 2 2 2
y 4|0 1 1 2 2 2 2 2
a 5|0 1 1 2 3 3 3 3
u 6|0 1 1 2 3 3 3 4
z 7|0 1 2 2 3 3 3 4
```

Implementaciju možemo izvršiti na sledeći način.

```
int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2 + 1, 0);

  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      if (s1[i-1] == s2[j-1])
        dp[i][j] = dp[i-1][j-1] + 1;
      else
        dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
    }

  return dp[n1][n2];
}
```

Prikažimo i kako je moguće rekonstruisati rezultujuću nisku na osnovu kompletne konstruisane matrice. Krećemo se od donjeg desnog ugla unazad i za svako polje proveravamo kako smo do njega došli (sa polja gore-levo od njega, sa polja levo od njega ili sa polja iznad njega) i na osnovu toga u rezultat dodajemo odgovarajući karakter.

```
string najduziZajednickiPodniz(const string& s1, const string& s2) {
  // konstrukcija matrice dp je ista kao u prethodnom rešenju
  ...

  string rez;
  rez.resize(dp[n1][n2]);
  int i = n1, j = n2, k = dp[n1][n2] - 1;
  while (k >= 0) {
```

```
    if (s1[i-1] == s2[j-1] && dp[i][j] == dp[i-1][j-1] + 1) {
      rez[k--] = s1[i-1];
      i--, j--;
    } else if (dp[i][j] == dp[i][j-1])
      j--;
    else
      i--;
  }
  return rez;
}
```

Ako nije neophodno rekonstruisati rešenje, već je dovoljno znati samo dužinu, kod možemo optimizovati tako što ne pamtimo čitavu matricu istovremeno. Naime, možemo primetiti da se prilikom popunjavanja matrice vrstu po vrstu sadržaj svake naredne vrste popunjava samo na osnovu prethodne vrste. Stoga nije potrebno istovremeno pamtiti celu matricu, već je dovoljno pamtiti samo jednu, tekuću vrstu. Ažuriranje vrste moramo vršiti s leva nadesno, jer svaki element u tekućoj vrsti zavisi od elementa koji mu prethodi u toj vrsti. Primetimo da nam je u nekom trenutku potrebno da znamo prethodni element tekuće vrste, a ponekad prethodni element prethodne vrste, tako da prilikom ažuriranja vrste moramo da u pomoćnoj promenljivoj pamtimo staru vrednost prethodnog elementa vrsta (jer se ažuriranjem prethodnog elementa njegova stara vrednost gubi, a ona nam može zatrebati u slučaju da su odgovarajući karakteri u niskama jednaki).

```
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];
}
```

Vremenska složenost ovog algoritma jednaka je $O(n_1 \cdot n_2)$, gde su n_1 i n_2 dužine date dve niske (ako su niske slične dužine, praktično je u pitanju algoritam kvadratne složenosti), a memorijska složenost je $O(n_2)$ (a lako se može spustiti do $O(\min(n_1, n_2))$, ako se pre primene algoritma niske urede tako da n_2 bude kraća od njih).

Najduža zajednička podniska

Problem: Defisati funkciju koja određuje dužinu najduže zajedničke podniske uzastopnih karaktera dve date niske. Na primer za niske `ababc` i `babbca` najduža zajednička podniska je `ab`.

Rešenje grubom silom podrazumevalo bi traženje svake podniske prve niske unutar druge niske i određivanje najduže podniske koja je pronađena i bilo bi vrlo neefikasno, čak i kada bi se primenjivali efikasni algoritmi traženja podniske.

Ovo je malo jednostavnija varijacija prethodnog zadatka. Pretpostavimo da je z najduža zajednička podniska niski x i y. Ako se odbace karakteri iza pojavljivanja niske z unutar x i karakteri iza pojavljivanja niske z unutar y, dobijaju se prefiksi reči x i y koji imaju z kao najduži zajednički sufiks. Za svaki par prefiksa dve date niske, dakle, potrebno je odrediti dužinu najvećeg sufiksa tih prefiksa na kom se oni poklapaju. Maksimum dužina takvih sufiksa za sve prefikse predstavljaće traženu dužinu najduže zajedničke podniske. Jedan način je da za svaki par prefiksa sufiks računamo iznova produžavajući ga na levo sve dok su završni karakteri tih prefiksa jednaki, no mnogo je bolje ako primetimo da je dužina najdužeg zajedničkog sufiksa jednaka nuli ako su poslednji karakteri dva prefiksa različiti, a da je za jedan veći od dužine najdužeg sufiksa dva prefiksa koja se dobijaju izbacivanjem poslednjih slova polazna dva prefiksa ako su poslednji karakteri dva prefiksa jednaki. Ako sa $f(m, n)$ označimo dužinu najdužeg sufiksa prefiksa reči a dužine m i prefiksa reči b dužine n, tada važi sledeća rekurentna veza.

$$
\begin{aligned}
f(m, 0) &= 0\\
f(0, n) &= 0\\
f(m, n) &= 1 + f(m-1, n-1), \quad \text{za } m, n > 0 \text{ i } a_{m-1} = b_{n-1}\\
f(m, n) &= 0, \quad \text{za } m, n > 0 \text{ i } a_{m-1} \neq b_{n-1}
\end{aligned}
$$

Ovo možemo pretočiti u rekurzivnu funkciju, koja će biti neefikasna zbog preklapanja rekurzivnih poziva. Ako primenimo tehniku dinamičkog programiranja odozdo naviše, dobijamo sledeću, efikasnu implementaciju (matricu popunjavamo vrstu po vrstu).

```
int najduzaZajednickaPodniska(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1 + 1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2 + 1, 0);

  int maxPodniska = 0;
  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
```

```
      if (s1[i-1] == s2[j-1])
        dp[i][j] = dp[i-1][j-1] + 1;
      else
        dp[i][j] = 0;
      if (dp[i][j] > maxPodniska)
        maxPodniska = dp[i][j];
    }

  return maxPodniska;
}
```

Prikažimo i primer matrice za niske `xyxy` i `yxyx`.

```
      y x y x
    0 1 2 3 4
   ----------
  0|0 0 0 0 0
x 1|0 0 1 0 1
y 2|0 1 0 2 0
x 3|0 0 2 0 3
y 4|0 1 0 3 0
```

Naravno, i ovde možemo upotrebiti memorijsku optimizaciju. Pošto element više ne zavisi od prethodnog elementa u tekućoj vrsti, elemente možemo ažurirati sdesna na levo.

```
int najduzaZajednickaPodniska(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2+1, 0);

  int maxPodniska = 0;
  for (int i = 1; i <= n1; i++)
    for (int j = n2; j >= 0; j--) {
      if (s1[i-1] == s2[j-1])
        dp[j] = dp[j-1] + 1;
      else
        dp[j] = 0;
      if (dp[j] > maxPodniska)
        maxPodniska = dp[j];
    }

  return maxPodniska;
}
```

Vremenska složenost ovog algoritma je $O(n_1 \cdot n_2)$, gde su n_1 i n_2 dužine niski, dok je memorijska složenost $O(n_2)$ (a eventualnom razmenom niski pre primene algoritma se može smanjiti na $O(\min(n1, n2))$).

Edit-rastojanje

Problem: Edit-rastojanje između dve niske se definiše u terminima operacija umetanja, brisanja i izmena slova prve reči kojima se može dobiti druga reč. Svaka od ove tri operacije ima svoju cenu. Definisati program koji izračunava najmanju cenu operacija kojima se od prve niske može dobiti druga. Na primer, ako je cena svake operacije jedinična, tada se niska `zdravo` može pretvoriti u `bravo!` najefikasnije operacijom izmene slova `z` u `b`, brisanja slova `d` i umetanja karaktera `!`.

Izvedimo prvo induktivno-rekurzivnu konstrukciju.

- Ako je prva niska prazna, najefikasniji način da se od nje dobije druga niska je da se umetne jedan po jedan karakter druge niske, tako da je minimalna cena jednaka proizvodu cene operacije umetanja i broja karaktera druge niske.
- Ako je druga niska prazna, najefikasniji način da se od prve niske dobije prazna je da se jedan po jedan njen karakter izbriše, tako da je minimalna cena jednaka proizvodu cene operacije brisanja i broja karaktera prve niske.
- Induktivna hipoteza će biti da umemo da rešimo problem za bilo koja dva prefiksa prve i druge niske. Ako su poslednja slova prve i druge niske jednaka, onda je potrebno pretvoriti prefiks bez poslednjeg slova prve niske u prefiks bez poslednjeg slova druge niske. Ako nisu, onda imamo tri mogućnosti. Jedna je da izmenimo jedan od ta dva karaktera u onaj drugi i onda da, kao u prethodnom slučaju, prevedemo prefikse bez poslednjih karaktera jedan u drugi. Druga mogućnost je da obrišemo poslednji karakter prve niske i probamo da pretvorimo tako njen dobijeni prefiks u drugu nisku. Treća mogućnost je da prvu nisku transformišemo u prefiks druge niske bez poslednjeg karaktera i da zatim dodamo poslednji karakter druge niske.

Na osnovu ovoga lako možemo definisati rekurzivnu funkciju koja izračunava edit-rastojanje. Da nam se niske ne bi menjale tokom rekurzije (što može biti sporo), efikasnije je da niske prosleđujemo u neizmenjenom obliku i da samo prosleđujemo brojeve karaktera njihovih prefiksa koji se trenutno razmatraju.

```
int editRastojanje(const string& s1, int n1,
                   const string& s2, int n2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  if (n1 == 0 && n2 == 0)
     return 0;
  if (n1 == 0)
     return n2 * cenaUmetanja;
  if (n2 == 0)
     return n1 * cenaBrisanja;
   if (s1[n1-1] == s2[n2-1])
     return editRastojanje(s1, n1-1, s2, n2-1);
```

```
  else {
    int r1 = editRastojanje(s1, n1-1, s2, n2) + cenaBrisanja;
    int r2 = editRastojanje(s1, n1, s2, n2-1) + cenaUmetanja;
    int r3 = editRastojanje(s1, n1-1, s2, n2-1) + cenaIzmene;
    return min({r1, r2, r3});
  }
}

int editRastojanje(const string s1&, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  return editRastojanje(s1, s1.size(), s2, s2.size(),
                        cenaUmetanja, cenaIzmene, cenaBrisanja);
}
```

Ovo rešenje je, naravno, neefikasno zbog preklapajućih rekurzivnih poziva. Algoritam dinamičkog programiranja za ovaj problem poznat je pod imenom Vagner-Fišerov algoritam. Rezultate za prefikse dužine i i j pamtićemo u matrici na polju (i, j). Dakle, ako su dužine niski n_1 i n_2, potrebna nam je matrica dimenzije $(n_1 + 1) \times (n_2 + 1)$, a konačan rezultat će se nalaziti na mestu (n_1, n_2). Za vežbu vam ostavljamo da implementirate rešenje tehnikom memoizacije. U nastavku ćemo prikazati rešenje pomoću dinamičkog programiranja naviše. Ako matricu popunjavamo vrstu po vrstu, sleva nadesno, prilikom izračunavanja elementa na poziciji (i, j), biće izračunati svi elementi matrice od kojeg on zavisi (a to su $(i-1, j-1)$, $(i-1, j)$ i $(i, j-1)$).

```
int editRastojanje(const string& s1, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2+1);

  dp[0][0] = 0;
  for (int i = 0; i <= n1; i++)
    dp[i][0] = i * cenaBrisanja;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      if (s1[i-1] == s2[j-1])
        dp[i][j] = dp[i-1][j-1];
      else {
        int r1 = dp[i-1][j] + cenaBrisanja;
        int r2 = dp[i][j-1] + cenaUmetanja;
        int r3 = dp[i-1][j-1] + cenaIzmene;
        dp[i][j] = min({r1, r2, r3});
      }
    }
```

```
  return dp[n1][n2];
}
```

Pod pretpostavkom da su cene jedinične, za niske `zdravo` i `bravo!` dobija se sledeća matrica.

```
     b r a v o !
   0 1 2 3 4 5 6
   -------------
 0|0 1 2 3 4 5 6
z1|1 1 2 3 4 5 6
d2|2 2 2 3 4 5 6
r3|3 3 2 3 4 5 6
a4|4 4 3 2 3 4 5
v5|5 5 4 3 2 3 4
o6|6 6 5 4 3 2 3
```

Na osnovu popunjene matrice lako je rekonstruisati i sam niz koraka koji prvu nisku transformiše u drugu. Krećemo od donjeg desnog ugla matrice i krećemo se unazad. U svakom koraku proveravamo kako smo došli na tekuću poziciju i u skladu sa tim korak ubacujemo u niz (vektor). Na kraju, kada stignemo do gornjeg levog ugla, niz koraka ispisujemo unazad. U tekućem primeru na polje (6, 6) smo stigli sa polja (6, 5) što znači da je poslednji korak umetanje karaktera `!`. Na polje (6, 5) smo stigli sa polja (5, 4) pri čemu nije vršen nikakva izmena. Slično, na polje (5, 4) smo stigli sa (4, 3), na polje (4, 3) smo stigli sa (3, 2), a na polje (3, 2) smo stigli sa (2, 1). Na polje (2, 1) smo mogli stići sa (1, 1) pri čemu je obrisan karaker `d`, a na polje (1, 1) smo stigli sa (0, 0) tako što je karakter `z` promenjen u `b`. Dakle, jedan niz mogućih koraka je

```
zdravo
          izmena z u b
bdravo
          brisanje d
bravo
          umetanje !
bravo!
```

Primetimo da smo na polje (2, 1) mogli stići i sa polja (1, 0) operacijom izmene `d` u `b`. Na polje (1, 0) smo stigli sa (0, 0) operacijom brisanja slova z. Na taj način dobijamo sledeći niz koraka.

```
zdravo
          brisanje z
dravo
          izmena d u b
bravo
          umetanje !
bravo!
```

Implementacija funkcije kojom se vrši rekonstrukcija (jednog) rešenja može biti sledeća.

```
void ispisiIzmene(const vector<vector<int>>& dp,
                  const string& s1, const string& s2,
                  int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  vector<string> izmene;
  int n1 = s1.size(), n2 = s2.size();
  while (n1 > 0 || n2 > 0) {
    if (n1 > 0 && n2 > 0 && s1[n1-1] == s2[n2-1] &&
        dp[n1][n2] == dp[n1-1][n2-1]) {
      n1--; n2--;
    } else if (n1 > 0 && n2 > 0 &&
               dp[n1][n2] == dp[n1-1][n2-1] + cenaIzmene) {
      izmene.push_back(string("Izmena: ") + s1[n1-1] + " -> " + s2[n2-1]);
      n1--; n2--;
    } else if (n2 > 0 && dp[n1][n2] == dp[n1][n2-1] + cenaUmetanja) {
      izmene.push_back(string("Umetanje: ") + s2[n2-1]);
      n2--;
    } else if (n1 > 0 && dp[n1][n2] == dp[n1-1][n2] + cenaBrisanja) {
      izmene.push_back(string("Brisanje: ") + s1[n1-1]);
      n1--;
    }
  }
  for (auto it = izmene.rbegin(); it != izmene.rend(); it++)
    cout << *it << endl;
}
```

Ponekad nam nisu bitne same izmene, već samo rastojanje (na primer, ako se vrši provera da li su dve niske bliske prilikom pretrage u kojoj se dopušta da je korisnik napravio i nekoliko slovnih grešaka). Pošto elementi tekućeg reda zavise samo od prethodnog, možemo izvršiti memorijsku optimizaciju i istovremeno čuvati samo jedan red. Tokom ažuriranja elementa na poziciji j njegov deo na pozicijama strogo manjim od j će čuvati elemente tekućeg reda i, deo od pozicije j nadalje će čuvati elemente prethodnog reda $i-1$. Promenjiva `prethodni` će čuvati vrednost sa polja $(i-1, j-1)$, a promenljiva `tekuci` će čuvati vrednost sa polja $(i-1, j)$.

```
int editRastojanje(const string& s1, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1);
  for (int j = 0; j <= n2; j++)
    dp[j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    dp[0] = i * cenaBrisanja;
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
```

```
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni;
      else {
        int r1 = tekuci + cenaBrisanja;
        int r2 = dp[j-1] + cenaUmetanja;
        int r3 = prethodni + cenaIzmene;
        dp[j] = min({r1, r2, r3});
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}
```

Najduži palindromski podniz

Problem: Napisati program koji određuje dužinu najdužeg palindromskog podniza date niske (podniz se dobija brisanjem karaktera polazne niske i čita se isto s leva na desno i s desna na levo). Na primer, za nisku `algoritmi_i_strukture_podataka` takav podniz je `at_rutur_ta`

Krenimo od rekurzivnog rešenja.

- Prazna niska ima samo prazan podniz, pa je dužina najdužeg palindromskog podniza jednaka nuli. Niska dužine 1 je sama svoj palindromski podniz, pa je dužina njenog najdužeg palindromskog podniza jednaka 1.
- Ako niska ima bar dva karaktera, onda ramzatramo da li su njen prvi i poslednji karakter jednaki. Ako jesu, onda oni mogu biti deo najdužeg palindromskog podniza i problem se svodi na pronalaženje najdužeg palindromskog podniza dela niske bez prvog i poslednjeg karaktera. U suprotnom oni ne mogu biti istovremeno biti deo najdužeg palindromskog podniza i potrebno je eliminisati bar jedan od njih. Problem, dakle, svodimo na pronalaženje najdužeg palindromskog podniza sufiksa niske bez prvog karaktera i na pronalaženje najdužeg palindromskog podniza prefiksa niske bez poslednjeg karaktera. Duži od ta dva palindromska podniza je traženi palindromski podniz cele niske.

Ovim je praktično definisana rekurzivna procedura kojom se rešava problem. U svakom rekurzivnom pozivu vrši se analiza nekog segmenta (niza uzastopnih karaktera polazne niske), pa je svaki rekurzivni poziv određen sa dva broja koji predstavljaju granice tog segmenta. Ako sa $f(l, d)$ označimo dužinu najdužeg palindromskog podniza dela niske $s[l, d]$, tada važe sledeće rekurentne veze.

$$f(l,d) = 0, \quad \text{za } l > d$$
$$f(l,d) = 1, \quad \text{za } l = d$$
$$f(l,d) = 1 + f(l+1, d-1), \quad \text{za } l < d \text{ i } s_l = s_d$$
$$f(l,d) = \max\left(f(l+1,d), f(l,d-1)\right), \quad \text{za } l < d \text{ i } s_l \neq s_d$$

Na osnovu ovoga, funkciju je veoma jednostavno implementirati.

```
int najduziPalindrom(const string& s, int p, int q) {
  if (p > q)
    return 0;
  if (p == q)
    return 1;
  if (s[p] == s[q])
    return 2 + najduziPalindrom(s, p+1, q-1);
  return max(najduziPalindrom(s, p, q-1),
             najduziPalindrom(s, p+1, q));
}

int najduziPalindrom(const string& s) {
  return najduziPalindrom(s, 0, s.length() - 1);
}
```

U prethodnoj funkciji dolazi do preklapanja rekurzivnih poziva, pa je poželjno upotrebiti memoizaciju. Za memoizaciju koristimo matricu (praktično, njen gornji trougao u kojem je $l < d$).

```
int najduziPalindrom(const string& s, int p, int q,
                     vector<vector<int>>& memo) {
  if (memo[p][q] != -1)
    return memo[p][q];
  if (p > q)
    return memo[p][q] = 0;
  if (p == q)
    return memo[p][q] = 1;
  if (s[p] == s[q])
    return memo[p][q] = 2 + najduziPalindrom(s, p+1, q-1, memo);
  return memo[p][q] = max(najduziPalindrom(s, p, q-1, memo),
                          najduziPalindrom(s, p+1, q, 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);
}
```

Do efikasnog rešenja možemo doći i dinamičkim programiranjem odozdo naviše. Element na poziciji (l,d) matrice zavisi od elemenata na pozicijama $(l+1,d)$,

$(l, d-1)$ i $(l+1, d-1)$, dok se konačno rešenje nalazi u gornjem levom uglu matrice, tj. na polju $(0, n-1)$. Zbog ovakvih zavisnosti matricu ne možemo popunjavati ni vrstu po vrstu, ni kolonu po kolonu, već dijagonalu po dijagonalu. Na dijagonalu ispod glavne upisujemo sve nule, na glavnu dijagonalu sve jedinice, a zatim popunjavamo jednu po jednu dijagonalnu iznad glavne, sve dok ne dođemo do elementa u gornjem levom uglu.

Prikažimo kako izgleda popunjena matrica na primeru niske `abaccba`.

```
    a b a c c b a
    0 1 2 3 4 5 6
    -------------
a 0|1 1 3 3 3 4 6
b 1|0 1 1 1 2 4 4
a 2|  0 1 1 2 2 4
c 3|    0 1 2 2 2
c 4|      0 1 1 1
b 5|        0 1 1
a 6|          0 1
```

Konačno rešenje 6 odgovara podnizu `abccba`.

```
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 r = 1; r < n; r++)
    for (int p = 0; p + r < n; p++) {
      int q = p + r;
      if (s[p] == s[q])
        dp[p][q] = dp[p+1][q-1] + 2;
      else
        dp[p][q] = max(dp[p+1][q], dp[p][q-1]);
    }

  return dp[0][n - 1];
}
```

Ovo rešenje ima i memorijsku i vremensku složenost $O(n^2)$.

Memorijsku složenost je moguće redukovati. Primećujemo da elementi svake dijagonale zavise samo od elemenata prethodne dve dijagonale. Moguće je da čuvamo samo dve dijagonale - tekuću i prethodnu. Tokom ažuriranja tekuće dijagonale njene postojeće elemente istovremeno prepisujemo u prethodnu. Kada su karakteri jednaki, tada u privremenu promenljivu beležimo odgovarajući element prethodne dijagonale, na njegovo mesto upisujemo odgovarajući element tekuće dijagonale, a onda na mesto tog elementa upisujemo vrednost privremene

promenljive uvećanu za dva. Kada su karakteri različiti odgovarajući element tekuće dijagonale upisujemo na odgovarajuće mesto u prethodnoj dijagonali, a na njegovo mesto upisujemo maksimum te i naredne vrednosti tekuće dijagonale.

```
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 r = 1; r < n; r++) {
    for (int p = 0; p + r < n; p++) {
      int q = p + r;
      if (s[p] == s[q]) {
        int tmp = dp[p];
        dp[p] = dpp[p+1] + 2;
        dpp[p] = tmp;
      }
      else {
        dpp[p] = dp[p];
        dp[p] = max(dp[p], dp[p+1]);
      }
    }
    dpp[n-r] = dp[n-r];
  }

  return dp[0];
}
```

Recimo još i da je rešenje ovog zadatka moguće dobiti i svođenjem na problem određivanja najdužeg zajedničkog podniza dve niske. Naime, najduži palindromski podniz jednak je najdužem zajedničkom podnizu originalne niske i niske koja se dobija njenim obrtanjem. Složenost ove redukcije je ista kao i složenost direktnog rešenja (vremenski $O(n^2)$, a prostorno $O(n)$).

Najduži rastući podniz

Problem: Napisati program koji određuje dužinu najdužeg strogo rastućeg podniza (ne obavezno uzastopnih elemenata) u datom nizu celih brojeva.

Recimo za početak da postoji veoma jednostavno svođenje ovog problema na problem pronalaženja najdužeg zajedničkog podniza dva niza, čije smo rešenje već prikazali. Naime, dužina najdužeg rastućeg podniza datog niza jednaka je dužini najdužeg zajedničkog podniza tog niza i niza koji se dobija neopadajućim sortiranjem i uklanjanjem duplikata tog niza. Ipak zadatak ćemo rešiti i direktno (pri tom, dobićemo i rešenje koje je efikasnije od kvadratnog, koje se dobija svođenjem na problem najdužeg zajedničkog podniza).

Zadatak rešavamo induktivno-rekurzivnom konstrukcijom, donekle slično

prethodnom primeru. Razmatraćemo poziciju po poziciju u nizu i odredićemo najduži rastući podniz čiji je poslednji element na svakoj od njih. - Najduži rastući podniz koji se završava na poziciji 0 je jednočlan niz a_0. - Prilikom određivanja dužine najdužeg rastućeg podniza koji se završava na poziciji $i > 0$, pretpostavićemo da za svaku prethodnu poziciju znamo dužinu najdužeg rastućeg podniza koji se na njoj završava. Niz koji se završava na poziciji i može produžiti sve one nizove koji se završavaju na nekoj poziciji $0 \leq j < i$ ako je $a_j < a_i$. Da bi niz koji se završava na poziciji j bio što duži, njegov prefiks koji se završava na poziciji j mora biti što duži (a dužine tih nizova možemo odrediti na osnovu induktivne hipoteze). Najduži od svih takvih nizova koje element a_i produžava će biti najduži niz koji se završava na poziciji i (ako ih nema, onda će najduži biti jednočlan niz a_i).

```
int najduziRastuciPodniz(const vector<int>& a, int i) {
   if (i == 0)
      return 1;
   int max = 1;
   for (int j = 0; j < i; j++) {
      int dj = najduziRastuciPodniz(a, j);
      if (a[i] > a[j] && dj + 1 > max)
          max = dj + 1;
   }
   return max;
}
```

Slično kao i kod Kadanovog algoritma, uz najduži niz koji se završava na poziciji i treba da znamo i najduži niz koji se završava na svim dosadašnjim pozicijama. Kada se iz prethodne funkcije dinamičkim programiranjem naviše uklone preklapajući rekurzivni pozivi, ta vrednost se može jednostavno odraditi naknadnim prolaskom kroz niz.

```
bool najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();

  vector<int> dp(n);
  dp[0] = 1;
  for (int i = 1; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++)
      if (a[i] > a[j] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;
  }

  int max = dp[0];
  for (int i = 0; i < n; i++)
    if (dp[i] > max)
      max = dp[i];
```

```
  return max;
}
```

Za razliku od Kadanovog algoritma gde svaki element u nizu zavisi samo od prethodnog u ovom slučaju element zavisi od svih prethodnih elemenata niza, pa se niz ne može zameniti sa jednom ili više promenljivih. Vremenska složenost ovog algoritma je $O(n^2)$, a memorijska složenost je $O(n)$.

Prikažimo i kako je moguće rekonstruisati neki rastući niz najveće dužine. Traženje maksimuma integrišemo u osnovnu petlju.

```
void najduziRastuciPodniz(const vector<int>& a, vector<int>& podniz) {
  int n = a.size();

  vector<int> dp(n);
  dp[0] = 1;
  int max = dp[0];
  for (int i = 1; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++)
      if (a[i] > a[j] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;
    if (dp[i] > max)
      max = dp[i];
  }

  // broj elemenata podniza
  podniz.resize(max);
  // podniz popunjavamo sa njegovog desnog kraja
  // k je prva nepopunjena pozicija (zdesna)
  int k = max - 1;
  // tražimo poslednju poziciju u nizu a na kojoj se završava
  // neki rastući podniz maksimalne dužine
  int i;
  for (i = n-1; dp[i] != max; i--)
    ;
  while (true) {
    // dodajemo element na kraj podniza
    podniz[k--] = a[i];
    // postupak se završava ako smo popunili ceo niz
    if (k < 0) break;
    // odredjujemo prethodnu poziciju na kojoj se završava
    // niz koji je produžen elementom a[i]
    for (int j = 0; j < i; j++) {
      if (a[i] > a[j] && dp[i] == dp[j] + 1) {
        i = j;
        break;
      }
    }
  }
```

```
}
```

Primetimo da je rešenja moglo biti više i da mi ovde konstruišemo samo jedno od njih. U nekim slučajevima mogu da nas zanimaju sva rešenja. Njih možemo rekonstruisati iscrpnom rekurzivnom pretragom.

```
void ispisiSveRastucePodnizove(vector<int>& podniz, int k, int i,
                               const vector<int>& a, const vector<int>& dp) {
  podniz[k--] = a[i];
  if (k < 0)
    ispisi(podniz);
  else
    for (int j = 0; j < i; j++)
      if (a[i] > a[j] && dp[i] == dp[j] + 1)
        ispisiSveRastucePodnizove(podniz, k, j, a, dp);
}

  vector<int> podniz(max);
  for (int i = 0; i < n; i++)
    if (dp[i] == max)
      ispisiSveRastucePodnizove(podniz, max-1, i, a, dp);
```

Efikasnije rešenje

Prethodno rešenje je složenosti $O(n^2)$, ali izmenom induktivno-rekurzivne konstrukcije možemo dobiti i mnogo efikasnije rešenje. Ključna ideja je da pretpostavimo da uz dužinu d_{max} najdužeg rastućeg podniza do sada obrađenog dela niza možemo da za svaku dužinu podniza $1 \leq d \leq d_{max}$ možemo da odredimo najmanji element kojim se završava rastući podniz dužine d. Primetimo da niz tih vrednosti uvek strogo rastući (ako postoji strogo rastući niz dužine d koji se završava nekim elementom a_i, tada se njegov prefiks dužine $d-1$ mora završavati nekim elementom koji je strogo manji od elementa a_i).

Niz obrađujemo element po element. Razmotrimo jedan primer (kolone tabele su označene dužinama niza), a elementi niza koji se obrađuju su napisani desno.

```
1 2 3 4 5 6 7 8 9 10
- - - - - - - - - -
3 - - - - - - - - -         3
2 - - - - - - - - -         2
2 6 - - - - - - - -         6
2 6 9 - - - - - - -         9
2 5 9 - - - - - - -         5
2 4 9 - - - - - - -         4
2 3 9 - - - - - - -         3
2 3 7 - - - - - - -         7
2 3 7 - - - - - - -         2
2 3 7 8 - - - - - -         8
```

Ako se dosadašnji najduži podniz završavao elementom koji je manji od tekućeg, onda smo našli podniz koji je duži za jedan i najmanji element na kraju tog podniza je tekući. To se u primeru dešava prilikom obrade elementa `3`, elementa `6`, elementa `9` i elementa `8`.

Razmotrimo situaciju u kojoj obrađujemo element `5`. Do tada smo videli elemente `3`, `2`, `6` i `9`. Element `2` na prvoj poziciji u tabeli označava da je najmanji element koji se može završiti jednočlani rastući niz jednak `2`. Element `6` na drugoj poziciji u tabeli označava da je najmanji element koji se može završiti dvočlani rastući niz jednak `6` (u pitanju je niz `2 6` ili niz `3 6`). Element `9` na trećoj poziciji u tabeli označava da je najmanji element koji se može završiti tročlani rastući niz jednak `9` (u pitanju je niz `2 6 9` ili `3 6 9`). Pošto je `5` manji od `9` nijedan od ovih tročlanih nizova nije moguće proširiti elementom `5`, pa je četvoročlanih rastućih nizova nema. Postavlja se pitanje da li se možda tročlani nizovi mogu završiti elementom `5`, no ni to nije moguće. Naime, pošto je u tabeli dvočlanim nizovima pridružena vrednost `6`, to znači da se svi dvočlani rastući nizovi završavaju bar sa `6`, pa nije moguće `9` zameniti sa `5`. Sa druge strane, pošto je `5` veće od `2`, završni element dvočlanih nizova `6` je moguće zameniti sa `5` i time dobiti manju završnu vrednost dvočlanih nizova (to su u ovom slučaju nizovi `3 5` i `2 5`). Dakle, u tabeli vrednost `6` treba zameniti vrednošću `5`. Vrednost `2` levo od `6` nema smisla zameniti sa `5`, jer bi se time završna vrednost jednočlanih nizova uvećala, a mi u tabeli pamtimo najmanje završne vrednosti.

Na osnovu analize ovog primera možemo da zaključimo da je prilikom analize svakog tekućeg elementa potrebno pronaći prvu poziciju d u tabeli na kojoj se nalazi element koji je veći ili jednak od tekućeg i poziciju d umesto toga upisati tekući element. Ako su svi elementi manji od tekućeg (ako je $d = d_{max}$), onda se tekući element dodaje na kraj niza (i u tom slučaju zapravo radimo isto - upisujemo element na poziciju d). Ostali elementi u tabeli ostaju nepromenjeni. Zaista na svim pozicijama u tabeli levo od pozicije d upisani su elementi strogo manji od tekućeg i njihovom zamenom sa tekućim se ne bi smanjila vrednost završnog elemenata tih nizova. Za elemente desno od pozicije d, iako su veći od tekućeg, ažuriranje nije moguće. U svim nizovima dužine $d' > d$ neki prefiks se morao završavati elementom na poziciji d ili elementom većim od njega, a pošto je on bio veći ili jednak od tekućeg, zamenog poslednjeg elementa tekućim ne bismo dobili više rastući niz.

Ključni dobitak nastaje kada se primeti da, pošto su elementi u tabeli sortirani, poziciju prvog elementa koji je veći ili jednak od tekućeg možemo ostvariti binarnom pretragom. Otuda sledi naredna efikasna implementacija (u nizu `dp` vrednost najmanjeg završnog elementa za nizove dužine d pamtimo na poziciji $d-1$).

```
int najduziRastuciPodniz(const vector<int>& a) {
  // broj elemenata niza a
  int n = a.size();
  // za svaku dužinu niza d na poziciji d-1 pamtimo vrednost najmanjeg
  // elementa kojim se može završiti podniz dužine d tekućeg dela niza
```

```
  vector<int> dp(n);
  // dužina najdužeg rastućeg podniza tekućeg dela niza
  int max = 0;
  // obrađujemo redom elemente niza
  for (int i = 0; i < n; i++) {
    // binarnom pretragom nalazimo poziciju prvog elementa u nizu dp
    // koji je veći ili jednak od tekućeg
    auto it = lower_bound(dp.begin(), next(dp.begin(), max), a[i]);
    int d = distance(dp.begin(), it);

    // element na toj poziciji menjamo tekućim
    dp[d] = a[i];

    // ako je izmenjen element na poziciji d, pronađen je niz dužine d+1
    // i u skladu sa tim ažuriramo maksimum
    if (d + 1 > max)
      max = d + 1;
  }

  return max;
}
```

Memorijska složenost je $O(n)$ i mogla bi se dodatno dovesti do $O(d_{max})$. Pošto se u svakom od n koraka vrši binarna pretraga dela niza dužine najviše n, složenost je $O(n \log n)$. Ova granica je asimptoski precizna, jer taj najgori slučaj zapravo nastupa u slučaju strogo rastućih nizova.

Iako se može pomisliti da je sadržaj tabele rastući podniz najveće dužine, to nije slučaj (u primeru koji smo razmatrali u trenutku kada u tabeli piše `2 5 9`, taj niz nije podniz polaznog niza i najduži rastući podniz je `2 6 9`).

Rekonstrukciju ne možemo efikasno izvršiti samo na osnovu poslednjeg reda tabele. Potrebno je da tokom implementacije pamtimo i niz dodatnih pomoćnih informacija. Jedna tehnička promena u implementaciji će biti to što nećemo pamtiti najmanje vrednosti završnih elemenata za svaku dužinu niza, već njihove pozicije (binarnu pretragu treba malo prilagoditi u skladu sa tim). Druga, suštinska će biti to što ćemo svakom elementu polaznog niza u trenutku njegovog upisivanja u tabelu pridružiti indeks poslednjeg elementa podniza koji se proširuje tekućim elementom da bi se dobio podniz čiji je poslednji element najmanji od svih mogućih poslednjih elemenata te dužine. To će tačno biti indeks koji se nalazi u tabeli pre njega.

```
void najduziRastuciPodniz(const vector<int>& a, vector<int>& podniz) {
  // dužina niza a
  int n = a.size();
  // za svaku dužinu niza d na poziciji d-1 pamtimo vrednost najmanjeg
  // elementa kojim se može završiti podniz dužine d tekućeg dela niza
  vector<int> dp(n);
  // pozicije prethodnih elemenata u rastućim nizovima
```

```
  vector<int> prethodni(n, -1);
  // dužina najdužeg rastućeg podniza tekućeg dela niza
  int max = 0;
  // obrađujemo redom elemente niza
  for (int i = 0; i < n; i++) {
    // binarnom pretragom nalazimo poziciju prvog elementa u nizu dp
    // koji je veći ili jednak od tekućeg
    auto it = lower_bound(begin(dp), next(begin(dp), max), i,
                          [a](int x, int y) {
                            return a[x] < a[y];
                          });
    int d = distance(dp.begin(), it);

    // element na toj poziciji menjamo tekućim
    dp[d] = i;

    // ako tekući element a[i] nastavlja neki neki rastući niz,
    // njegovoj poziciji i pridružujemo poziciju prethodnog
    // elementa u tom rastučem nizu
    if (d > 0)
      prethodni[i] = dp[d-1];

    // ako je izmenjen element na poziciji d, pronađen je niz dužine d+1
    // i u skladu sa tim ažuriramo maksimum
    if (d + 1 > max)
      max = d + 1;
  }

  // alociramo memoriju za elemente podniza
  podniz.resize(max);
  // popunjavamo elemente podniza prateći linkove ka prethodnim elementima
  // unazad
  for (int k = max-1, i = dp[k]; k >= 0; k--, i = prethodni[i])
    podniz[k] = a[i];
}
```

Najveći kvadrat u matrici

Problem: Data je pragougaona matrica koja sadrži samo nule i jedinice. Napiši program koji određuje dimenziju najvećeg kvadratnog bloka koji se sastoji samo od jedinica.

Osnovna ideja rešenja je da za svako polje (i, j) izračunamo dimenziju najvećeg kvadrata čije je donje desno teme na tom polju. To možemo uraditi induktivno-rekurzivno.

- Za polja u prvoj vrsti i prvoj koloni matrice, ti kvadrati su dimenzije ili 0 ili 1 u zavisnosti od elementa matrice na tom polju.

- Za polja u unutrašnjosti matrice na kojima se nalazi 0, takav kvadrat ne postoji tj. dimenzija mu je 0. Za polja na kojima se nalazi 1, rešenje zavisi od dimenzija kvadrata čija su temena na poljima $(i-1, j-1)$, $(i-1, j)$ i $(i, j-1)$. Neka je najmanja dimenzija ta tri kvadrata m. Tvrdimo da je dimenzija kvadrata sa donjim levim temenom na (i, j) jednaka $m+1$. Dokažimo ovo.

  Ako je minimalni od ta tri kvadrata onaj sa donjim desnim temenom na polju $(i-1, j-1)$, tada kvadrati sa donjim desnim temenima na poljima $(i, j-1)$ i $(i-1, j)$ imaju dimenziju bar m. To znači da je bar m elemenata u vrsti i levo od polja (i, j) i da je bar m elemenata u koloni j iznad polja (i, j) popunjeno jedinicama. Pošto je i na polju (i, j) jedinica, znači da postoji kvadrat čija je dimenzija bar $m+1$ čije je donje desno teme na polju (i, j).

  Slično, ako je minimalni od ta tri kvadrata onaj sa donjim desnim temenom na polju $(i, j-1)$, tada kvadrat sa donjim desnim temenom na polju $(i-1, j)$ ima dimenziju bar m. Na osnovu prvog znamo su elementi u bar m vrsta iznad vrste i u koloni j jednaki 1. Pošto je i na polju (i, j) jedinica, znači da postoji kvadrat čija je dimenzija bar $m+1$ čije je donje desno teme na polju (i, j).

  I u slučaju kada je minimalni kvadrat onaj sa donjim desnim temenom na poplju $(j-1, i)$ potpuno analogno se dokazuje da postoji kvadrat dimenzije bar $m+1$ čije je donje desno teme na polju (i, j).

  Nije moguće da postoji kvadrat sa donjim desnim temenom na polju (i, j) čija bi dimenzija bila $m' > m+1$. Naime, ako bi takav kvadrat postojao, on bi obuhvatao kvadrate sa donjim desnim temenim na poljima $(i-1, j)$, $(i, j-1)$ i $(i-1, j-1)$ čija bi dimenzija bila za jedan manja od m', pa samim tim strogo veća od m. To je u jasnoj kontradikciji da najmanji od ta tri maksimalna kvadrata ima dimenziju m.

Na osnovu prethodne diskusije, rekurzivna implementacija sledi direktno.

```
int maxKvadrat(int A[MAX][MAX], int m, int n, int i, int j) {
    if (i == 0 || j == 0)
       return A[i][j];

    if (A[i][j] == 0)
       return 0;
    int m1 = maxKvadrat(A, m, n, i-1, j-1);
    int m2 = maxKvadrat(A, m, n, i, j-1);
    int m3 = maxKvadrat(A, m, n, i-1, j);
    return min({m1, m2, m3}) + 1;
}

int maxKvadrat(int A[MAX][MAX], int m, int n) {
  int max = 0;
  for (int i = 0; i < m; i++)
```

```
    for (int j = 0; j < n; j++) {
      int m = maxKvadrat(A, m, n, i, j);
      if (m > max)
        max = m;
    }
  return max;
}
```

Međutim ovde jasno dolazi do preklapajućih poziva i funkcija je neefikasna. Takođe, nama je potrebno da znamo maksimalnu dimenziju svih ovakvih kvadrata (jer je naš traženi maksimalni kvadrat jedan od njih), tako da je potrebno znati rezultat svih rekurzivnih poziva za $0 \leq i < m$ i $0 \leq j < n$. Oba se problema razrešavaju ako koristimo dinamičko programiranje naviše. Ako matricu popunjavamo vrstu po vrstu, sleva nadesno, prilikom izračunavanja svakog elementa na poziciji (i, j) biće već izračunata sva tri elementa od kojih on zavisi.

```
int maxKvadrat(int A[MAX][MAX], int m, int n) {
  int dp[MAX][MAX];
  for (int i = 0; i < m; i++)
    dp[i][0] = A[i][0];
  for (int j = 0; j < n; j++)
    dp[0][j] = A[0][j];
  for (int i = 1; i < m; i++)
    for (int j = 1; j < n; j++)
      if (A[i][j] == 0)
        dp[i][j] = 0;
      else
        dp[i][j] = min({dp[i-1][j-1], dp[i][j-1], dp[i-1][j]}) + 1;
  int max = 0;
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      if (dp[i][j] > max)
        max = dp[i][j];
  return max;
}
```

Pošto elementi svake vrste zavise samo od elemenata prethodne vrste, moguće je opet izvršiti memorijsku optimizaciju.

```
int maxKvadrat(int A[MAX][MAX], int m, int n) {
  int dp[MAX];
  int max = 0;
  for (int j = 0; j < n; j++) {
    dp[j] = A[0][j];
    if (dp[j] > max)
      max = dp[j];
  }
```

```
  for (int i = 1; i < m; i++) {
     int prethodni = dp[0];
     dp[0] = A[i][0];
     if (dp[0] > max)
       max = dp[0];
     for (int j = 1; j < n; j++) {
        int tekuci = dp[j];
        if (A[i][j] == 0)
           dp[j] = 0;
        else
           dp[j] = min({prethodni, tekuci, dp[j-1]}) + 1;
        if (dp[j] > max)
           max = dp[j];
        prethodni = tekuci;
     }
  }
  return max;
}
```

Optimalni raspored zagrada

Problem: Množenje matrica dimenzije $D_1 \times D_2$ i dimenzije $D_2 \times D_3$ daje matricu dimenzije $D_1 \times D_3$ i da bi se ono sprovelo potrebno je $D_1 \cdot D_2 \cdot D_3$ množenja brojeva. Kada je potrebno izmnožiti duži niz matrica, onda efikasnost zavisi od načina kako se te matrice grupišu (množenje je asocijativna operacija i dopušten je bilo koje grupisanje pre množenja. Napiši program koji za dati niz brojeva $D_0, D_1, D_2, \ldots D_{n-1}$ određuje minimalni broj množenja brojeva prilikom množenja matrica dimenzija $D_0 \times D_1$, $D_1 \times D_2$, $\ldots D_{n-2} \times D_{n-1}$.

Krenimo od rekurzivnog rešenja. Kako god da grupišemo matrice neko množenje je to koje se poslednje izvršava. To biti množenje prve matrice i proizvoda svih ostalih, množenje proizvoda prve dve matrice i proizvoda svih ostalih matrica, množenje proizvoda prve tri matrice i proizvoda ostalih matrica itd. sve do množenja proizvoda svih matrica pre poslednje poslednjom matricom. Osnovna ideja je da ekplicitno izanaliziramo sve te mogućnosti i da odaberemo najbolju od njih. Za svaki fiksirani izbor pozicije poslednjeg množenja potrebno je odrediti kako množiti sve matrice levo i sve matrice desno od te pozicije. Ključni uvid je da je da bi globalni izbor bio optimalan i ta dva potproblema potrebno rešiti na optimalan način (jer u slučaju da ne odaberemo optimalni raspored zagrada u nekom potproblemu, bolje globalno rešenje možemo dobiti zamenog tog rasporeda optimalnim). Dakle, potprobleme možemo rešavati rekurzivnim pozivima.

Neka $f(l, d)$ označava minimalni broj množenja potreban da se izmnože matrice dimenzija $D_l, \ldots, D_d$. Potrebno je odrediti $f(0, n-1)$. Na osnovu prethodne diskusije, važe sledeće rekurentne veze.

$$f(l,d) = 0, \quad \text{za } d-l+1 \leq 2$$
$$f(l,d) = \min_{l<i<d} (f(l,i) + f(i,d) + D_l \cdot D_i \cdot D_d), \quad \text{za } d-l+1 \geq 3$$

Zaista, pozicija poslednjeg množenja može biti svaka pozicija strogo veća od l i strogo manja od d. Kada je poslednje množenje na poziciji i znači da se na kraju množi proizvod matrica dimenzija $D_l, \ldots, D_i$ i proizvod matrica dimenzija $D_i, \ldots, D_d$. Rekurzivno određujemo minimalni broj množenja za svaki od tih proizvoda. Prvi proizvod daje matricu dimenzije $D_l \times D_i$, drugi daje matricu dimenzije $D_i \times D_d$, i na kraju se vrši množenje te dve matrice, za šta je potrebno dodatnih $D_l \times D_i \times D_d$ operacija.

```
int minBrojMnozenja(const vector<int>& dimenzije, int l, int d) {
  int n = d - l + 1;
  if (n <= 2)
    return 0;
  int min = numeric_limits<int>::max();
  for (int i = l+1; i <= d-1; i++) {
    int broj = minBrojMnozenja(dimenzije, l, i) +
               minBrojMnozenja(dimenzije, i, d) +
               dimenzije[l] * dimenzije[i] * dimenzije[d];
    if (broj < min)
      min = broj;
  }
  return min;
}

int minBrojMnozenja(const vector<int>& dimenzije) {
  return minBrojMnozenja(dimenzije, 0, dimenzije.size() - 1);
}
```

Direktno rekurzivno rešenje dovodi do veliki broj identičnih rekurzivnih poziva i neuporedivo bolje rešenje se dobija dinamičkim programiranjem. Najjednostavnije rešenje je dodati memoizaciju rekurzivnoj funkciji. Pošto funkcija ima dva promenljiva celobrojna parametra, memoizaciju možemo izvršiti pomoću matrice dimenzije $n \times n$. Konačno rešenje se nalazi na poziciji $(0, n-1)$.

```
int minBrojMnozenja(const vector<int>& dimenzije, int l, int d, vector<vector<int>>& memo) {
  if (memo[l][d] != -1)
    return memo[l][d];
  int n = d - l + 1;
  if (n == 2)
    return 0;
  int min = numeric_limits<int>::max();
  for (int i = l+1; i <= d-1; i++) {
    int broj = minBrojMnozenja(dimenzije, l, i, memo) +
                     minBrojMnozenja(dimenzije, i, d, memo) +
```

```
                  dimenzije[l] * dimenzije[i] * dimenzije[d];
    if (broj < min)
      min = broj;
  }
  return memo[l][d] = min;
}

int minBrojMnozenja(const vector<int>& dimenzije) {
  int n = dimenzije.size();
  vector<vector<int>> memo(n);
  for (int i = 0; i < n; i++)
    memo[i].resize(n, -1);
  return minBrojMnozenja(dimenzije, 0, dimenzije.size() - 1, memo);
}
```

Dinamičko programiranje naviše je u ovom slučaju komplikovanije, zbog komplikovanijih zavisnosti između elemenata matrice. Prvo, pošto važi da je $l \leq d$, relevantan nam je samo deo matrice iznad njene glavne dijagonale. Popunjavanje nije moguće ni po vrstama, ni po kolonama. Može se uočiti da svaki element zavisi samo od elemenata koji se nalaze ispod dijagonale na kojoj se taj element nalazi. Zato je elemente moguće izračunavati po dijagonalama. Elemente na glavnoj dijagonali i dijagonali inzad nje postavljamo na nulu, a zatim računamo elemente na dijagonalama iznad njih. Svaku dijagonalu karakteriše konstantni razmak između indeksa l i d tj. isti broj elemenata u intervalu $[l, d]$ tj. isti broj matrica koje se množe.

```
int minBrojMnozenja(const vector<int>& dimenzije) {
  int n = dimenzije.size();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n, 0);

  for (int k = 3; k <= n; k++)
    for (int l = 0, d = l + k - 1; d < n; l++, d++) {
      dp[l][d] = numeric_limits<int>::max();
      for (int i = l+1; i <= d-1; i++) {
        int broj = dp[l][i] + dp[i][d] +
          dimenzije[l] * dimenzije[i] * dimenzije[d];
        if (broj < dp[l][d])
          dp[l][d] = broj;
      }
    }
  return dp[0][n-1];
}
```

Prikažimo na jednom primeru kako se gradi i popunjava matrica. Neka su matrice dimenzija 4, 3, 5, 1, 2.

```
   0  1  2  3  4
```

```
-------------------
0 |  0  0 60 27 35
1 |  0  0  0 15 21
2 |  0  0  0  0 10
3 |  0  0  0  0  0
4 |  0  0  0  0  0
```

Analizom zavisnosti između elemenata matrice može se ustanoviti da nije moguće jednostavno smanjivanje memorijske složenosti, kao što je to bio slučaj u nekim ranije prikazanim problemima.

Naravno, osim vrednosti optimuma, želimo da dobijemo i efektivni način da brzo pomnožimo matrice. Potrebno je, dakle, da izvršimo i rekonstrukciju rešenja. Iako bismo prilikom rekonstrukcije u svakom koraku na osnovu same matrice dinamičkog programiranja mogli da proveravamo na kojoj poziciji se nalazi poslednje množenje koje se vrši, implementacija je malo jednostavnija ako te vrednosti registrujemo u posebnoj matrici (po cenu dodatnog utroška memorije). Ako imamo tu matricu na raspolaganju, optimalni raspored zagrada možemo ispisati narednom jednostavnom rekurzivnom procedurom.

```
void odstampaj(const vector<vector<int>>& pozicija, int l, int d) {
  int n = d - l + 1;
  if (n <= 2)
    cout << "A" << l;
  else {
    cout << "(";
    odstampaj(pozicija, l, pozicija[l][d]);
    cout << "*";
    odstampaj(pozicija, pozicija[l][d], d);
    cout << ")";
  }
}
```

Najveći pokupljeni zbir

Problem: Matrica sadrži prirodne brojeve. Koliki se najveći zbir može postići ako se kreće iz gornjeg levog ugla i u svakom koraku se kreće ili na susedno polje desno, ili na susedno polje dole.

Rešenje možemo ostvariti grubom silom, tj. proverom svih mogućih putanja. Naredna funkcija određuje najveći zbir koji se može postići ako se sa polja (v, k) stiže na polje $(n-1, n-1)$ u matrici M dimenzije n. Ako je početno polje jednako ciljnom, tada se može pokupiti samo broj sa tom polju. Ako polje u poslednjoj vrsti, tada se do poslednjeg polja stiže samo preko polja desno od njega, a ako je u poslednjoj koloni, onda se stiže samo polja ispod njega. U suprotnom se bira da li će se do kraja stići praveći korak desno ili korak dole. U svim tim slučajevima rekurzivno računamo optimum polja na koje smo prešli (birajući bolju od dve mogućnosti, ako nismo na rubu matrice). Naglasimo da je

ovde ispunjen uslov optimalne podstrukture, jer kada bismo od polja na koje smo prešli imali neki neoptimalni put, mogli bismo ga zameniti optimalni i time popraviti rešenje za polje sa kog smo krenuli.

```
int maksZbir(const vector<vector<int>>& M, int n, int v, int k) {
    if (v == n-1 && k == n-1)
        return M[v][k];
    int dole, desno;
    if (v < n - 1)
       dole = M[v][k] + maksZbir(M, n, v+1, k);
    if (k < n - 1)
       desno = M[v][k] + maksZbir(M, n, v, k+1);
    if (v == n-1) return desno;
    if (k == n-1) return dole;
    return max(dole, desno);
}

int maksZbir(const vector<vector<int>>& M) {
   int n = M.size();
   return maksZbir(M, n, 0, 0);
}
```

Drugi način je da za svako polje određujemo koliki je najveći zbir koji se može pokupiti od polja $(0, 0)$ do tog polja. Razmatranje je veoma slično kao u prethodnom slučaju.

```
int maksZbir(const vector<vector<int>>& M, int n, int v, int k) {
    if (v == 0 && k == 0)
        return M[v][k];
    int odGore, sLeva;
    if (v > 0)
       odGore = M[v][k] + maksZbir(M, n, v-1, k);
    if (k > 0)
       sLeva = M[v][k] + maksZbir(M, n, v, k-1);
    if (v == 0) return sLeva;
    if (k == 0) return odGore;
    return max(odGore, sLeva);
}

int maksZbir(const vector<vector<int>>& M) {
   int n = M.size();
   return maksZbir(M, n, n-1, n-1);
}
```

U obe prethodne rekurzivne definicije postoji veliki broj preklapajućih rekurzivnih poziva i potrebno je izvršiti optimizaciju dinamičkim programiranjem. Možemo uvesti matricu u kojoj ćemo za svako polje pamtiti najveći zbir koji se može ostvariti krećući se od početnog polja do tog polja.

Pretpostavimo da je ulazna matrica jednaka

```
4 3 1 1 1
1 9 2 1 3
1 3 5 1 2
1 3 1 2 0
4 6 7 2 1
```

Tada je matrica dinamičkog programiranja u drugom rešenju jednaka

```
 4  7  8  9 10
 5 16 18 19 22
 6 19 24 25 27
 7 22 25 27 27
11 28 35 37 38
```

Najveći mogući zbir koji se može pokupiti je 38.

Ponovo možemo izvršiti memorijsku optimizaciju. Popunjavamo vrstu po vrstu i čuvamo samo tekuću vrstu. Ažuriranja vrste moramo vršiti sa leva nadesno, što je u redu, jer se pri računanju svake sledeće vrednosti koristi stara vrednost u tekućoj koloni i nova vrednost u prethodnoj koloni.

```
int maksZbir(const vector<vector<int>>& M) {
  int n = M.size();
  vector<int> dp(n);
  dp[0] = M[0][0];
  for (int k = 1; k < n; k++)
    dp[k] = dp[k-1] + M[0][k];
  for (int v = 1; v < n; v++) {
    dp[0] += M[v][0];
    for (int k = 1; k < n; k++)
      dp[k] = max(dp[k] + M[v][k], dp[k-1] + M[v][k]);
  }
  return dp[n-1];
}
```

0-1 problem ranca

Problem: Dato je n predmeta čije su mase celi brojevi $m_0, \ldots m_{n-1}$ i cene realni brojevi $c_0, \ldots, c_{n-1}$. Napisati program koji određuje naveću cenu koja se može pokupiti pomoću ranca celobrojne nosivosti M (nije moguće uzimati delove predmeta, niti isti iste predmete više puta).

Krenimo od rekurzivnog rešenja kakvo smo implementirali u sklopu proučavanja pretrage i odsecanja. Funkcija vraća najveću cenu koja se može postići za ranac date nosivosti ako se posmatra samo prvih n predmeta. Rešenje je veoma jednostavno i zasnovano na induktivno-rekurzivnom pristupu. Bazu čini slučaj $n = 0$, kada je maksimalna moguća cena jednaka nuli jer nemamo predmeta koje bismo uzimali. Kada je $n > 0$ izdvajamo poslednji predmet i razmatramo mogućnost da on nije ubačen i da jeste ubačen u ranac. Drugi slučaj je moguć

samo ako je masa tog predmeta manja ili jednaka od nosivosti ranca. Veća od te dve cene predstavlja optimalnu cenu. Rekurzivnim pozivima se traži optimum za skup bez poslednjeg predmeta, što je u redu, jer je za globalni optimum neophodno i da su predmeti iz tog podskupa odabrani optimalno (zadovoljen je uslov optimalne podstrukture).

```
double maxCena(const vector<int>& mase, const vector<double>& cene,
               double nosivost, int n) {
  if (n == 0)
     return 0.0;
  double cenaBez = maxCena(mase, cene, nosivost, n-1);
  if (mase[n-1] > nosivost)
     return cenaBez;
  double cenaSa = maxCena(mase, cene, nosivost - mase[n-1], n-1) +
                  cene[n-1];
  return max(cenaBez, cenaSa);
}
```

U ovoj implementaciji sasvim je moguće da se identični rekurzivni pozivi ponove više puta, što dovodi do neefikasnosti. Rešenje, naravno, dolazi u obliku dinamičkog programiranja (bilo memoizacije, bilo dinamičkog programiranja naviše). Pošto imamo dva promenljiva parametra, alociramo takvu matricu da svakoj vrsti odgovara jedan prefiks niza predmeta, a svakoj koloni jedna nosivost. Matricu možemo popunjavati vrstu po vrstu. Takođe, možemo primetiti da elementi svake vrste zavise samo od prethodne, tako da ne moramo čuvati celu matricu, već samo tekuću vrstu. Ažuriranje vršimo s desnog kraja.

```
double maxCena(const vector<int>& mase, const vector<double>& cene,
               double nosivost, int n) {
  vector<double> dp(nosivost + 1);
  dp[0] = 0.0;
  for (int N = 1; N <= n; N++) {
    for (int M = nosivost; M >= 0; M--)
      if (mase[N-1] <= M)
        dp[M] = max(dp[M], dp[M - mase[N-1]] + cene[N-1]);
  }
  return dp[nosivost];
}
```

Ovim smo dobili algoritam čija je memorijska složenost $O(M)$ gde je M nosivost ranca, dok je velika složenost $O(N \cdot M)$ gde je N broj predmeta, a M nosivost ranca. Obratimo pažnju na to da iako deluje da smo ovaj problem rešili u polinomijalnoj složenosti, to zapravo nije slučaj. Naime, složenost nije izražena samo u terminima veličine ulaza, već u terminu vrednosti na ulazu (vrednosti nosivosti ranca). Za ovakve algoritme se kaže da su pseudo-polinomijalni. Veličina ulaza vezanog za broj M odgovara broju cifara broja M (npr. broju binarnih cifara upotrebljenih u zapisu), a vreme izvršavanja algoritma eksponencijalno raste u odnosu na taj broj.

Optimalno pakovanje

Problem: Ispred lifta čeka $n \leq 20$ matematičara. Poznata je masa svakog od njih i ukupna nosivost lifta (svako pojedinačno može da stane u lift). Odrediti minimalan broj vožnji potreban da se svi prevezu.

Ovaj problem je u teoriji poznat kao *jednodimenzionalni problem pakovanja* (engl. 1D bin packing problem). Rešenje grubom silom bi podrazumevalo da se razmotre sve njihove moguće podele u vožnje. Jedan način da se to uradi je da se razmotre svi njihovi mogući redosledi ulaska u lift i da se onda za svaki fiksirani redosled oni pakuju u lift jedan po jedan, dok god je to moguće i tako odredi potreban broj vožnji. Složenost tog pristupa bio bi $O(n!)$ i teško da bi mogao da se primeni na rešavanje instanci za $n \geq 15$ u nekom realnom vremenu (u sekundama).

```
int brojVoznji(const vector<int>& tezine, int nosivost) {
  // ukupan broj osoba
  int n = tezine.size();

  // osobe se sigurno mogu prevesti u n vožnji
  int minBrojVoznji = n;

  // krećemo od rasporeda 1, ..., n
  vector<int> permutacija(n);
  for (int i = 0; i < n; i++)
    permutacija[i] = i;

  // obrađujemo sve moguće rasporede
  do {
    // potreban broj vožnji
    int brojVoznji = 1;
    // masa ljudi u liftu u trenutnoj vožnji
    int uLiftu = 0;
    // redom obrađujemo sve matematičare po trenutnom rasporedu
    for (int i = 0; i < tezine.size(); i++) {
      // ako trenutni ne može da stane u lift,
      // otvaramo novu vožnju
      if (uLiftu + tezine[permutacija[i]] > nosivost) {
        uLiftu = 0;
        brojVoznji++;
      }
      // dodajemo trenutnog u lift
      uLiftu += tezine[permutacija[i]];
    }
    // ažuriramo minimalan broj vožnji
    if (brojVoznji < minBrojVoznji)
      minBrojVoznji = brojVoznji;

    // prelazimo na narednu permutaciju
```

```
  } while (next_permutation(begin(permutacija), end(permutacija)));

  // vraćamo minimalni rezultat
  return minBrojVoznji;
}
```

Primetimo da smo permutacije jednostavno generisali korišćenjem bibliotečke funkcije `next_permutation`.

Postoje i načini da se problem reši približno. Odokativna procena potrebnog broja vožnji je količnik ukupne mase svih i nosivosti lifta, zaokružen naviše. To je zapravo donja granica i broj vožnji ne može biti manji od toga i on se dobija ako su liftovi uvek maksimalno popunjeni, što nije realno očekivati (npr. ako je nosivost 100, a svi imaju po 51 kilogram, broj vožnji će biti mnogo veći nego ova donja granica).

Bolji način od toga je heuristika u kojoj bi se u lift ubacivala najteža osoba koja može da stane u njega i nova vožnja pokretala tek kada nijedna od preostalih osoba ne može da stane u tekući lift. Ta heuristika daje prilično bliska rešenja tačnom, ali se ne može garantovati da će uvek dati tačno rešenje. Na primer, ako je nosivost lifta 21, a mase su 7, 5, 7, 5, 9, 10, 9, i 9 jedinica, ovim gramzivim pristupom bi se putnici rasporedili u vožnje od $10 + 9$, $9 + 9$, $7 + 7 + 5$ i 5, tako da bi bilo ukupno četiri vožnje. Optimalno rešenje bi rasporedilo putnike u tri vožnje $9 + 7 + 5$, $9 + 7 + 5$, i $10 + 9$.

Implementacija može biti sledeća.

```
int brojVoznji(vector<int>& tezine, int nosivost) {
  int n = tezine.size();
  // sortiramo po težini nerastuće
  sort(begin(tezine), end(tezine), greater<int>());
  // beležimo da li se osoba i već odvezla
  vector<bool> odvezaoSe(n, false);
  // broj preostalih osoba koje čekaju lift
  int preostalo = n;
  // trenutna masa ljudi u liftu
  int uLiftu = 0;
  // potreban broj vožnji
  int broj = 1;
  // dok se nisu svi prevezli
  while (preostalo > 0) {
    // tražimo najtežeg od preostalih koji može da stane u lift
    int i;
    for (i = 0; i < n; i++)
      if (!odvezaoSe[i] && uLiftu + tezine[i] <= nosivost) {
        // osoba i ulazi u lift
        odvezaoSe[i] = true;
        preostalo--;
        uLiftu += tezine[i];
        break;
```

```
    }
    // ako niko od preostalih nije mogao da stane u lift
    // lift odlazi i pokrećemo novu vožnju
    if (i == n) {
      broj++;
      uLiftu = 0;
    }
  }
  return broj;
}
```

Složenost ovog rešenja je $O(n^2)$ i može se boljom implementacijom još unapred-iti, što je neuporedivo bolje od potpuno tačnih rešenja, ali nam ne garantuje ispravnost. Ipak, može se pokazati da broj vožnji dobijen ovakvom heuristikom nije gori od optimuma za više od dvadesetak procenata.

Prikazaćemo rešenje zasnovano na dinamičkom programiranju koje će nam omogućiti da uspešno pronađemo garantovano optimalna rešenja za vrednosti n oko 20 (postoje i bolji algoritmi, koji omogućavaju da se egzaktno reše problemi i za n oko 100, ali oni su dosta kompleksniji). Osnovna ideja rešenja se zasniva na tome da je moguće ojačati induktivnu hipotezu i uz minimalni broj vožnji vratiti i težinu u najmanje opterećenoj vožnji. Možemo uvek pretpostaviti da je to poslednja vožnja (jer ako nije, možemo ljude iz poslednje vožnje zameniti sa ljudima iz te najmanje opterećene vožnje).

- Baza indukcije može biti slučaj kada imamo samo jednu osobu i u tom slučaju je optimalno rešenje jedna vožnja. Ta vožnja je najmanje popunjena od svih (jer drugih nema) i u njoj je masa jednaka masi te jedne osobe.
- Pretpostavimo da umemo da rešimo problem za svaki skup koji ima manje od n osoba i razmotrimo kako bismo mogli da ga rešimo za n osoba. Jedna od tih n osoba će biti ona koja se vozi poslednja. Ne znamo koji izbor vodi do najboljeg rešenja, tako da moramo da isprobamo sve moguće izbore te poslednje osobe. Za svaki takav izbor, rešimo rekurzivno problem za sve osobe bez te i dobijemo najmanji broj vožnji i masu osoba u poslednjoj, najmanje opterećenoj vožnji. Ako osoba staje u lift u toj poslednjoj vožnji, onda znamo da je broj vožnji sa njom jednak broj vožnji bez nje. U suprotnom, znamo da broj vožnji mora biti za jedan veći (jer ta osoba ne može stati ni u jednu prethodnu vožnju, jer je poslednja vožnja najmanje opterećena). Najmanji broj vožnji za n osoba dobijamo kao minimum potrebnih brojeva vožnji za svaki izbor poslednje osobe. Ostaje još pitanje kako je moguće odrediti najmanji moguću opterećenost poslednje vožnje pri tom optimalnom broju vožnji i ovde treba pažljivo razmisliti.

  Naime, ako osoba staje u lift u poslednjoj vožnji koja je bila najmanje opterećena, ne znači da će nakon njenog ulaska ta poslednja vožnja biti i dalje najmanje opterećena. Takođe i kada uđe sama u lift ne znači da će ta poslednja vožnja biti najmanje opterećena (jer je možda ta osoba teža od svih osoba u prethodnoj vožnji). Na primer, ako imamo osobe težina 6, 4 i

3, ako je kapacitet lifta 8 i ako pretpostavimo da je osoba 3 poslednja osoba, tada se rekurzivnim pozivom dobija da su za prevoz osoba 6 i 4 potrebne dve vožnje i da u najmanje opterećenoj poslednjoj vožnji može biti osoba 4. Dodavanjem osobe 4 u poslednju vožnju dobijamo 7, pa znamo da je su i za sve tri osobe dovoljne dve vožnje, ali greška bi bila da zaključimo da je 7 najmanji mogući kapacitet u poslednjoj vožnji. Do boljeg rešenja se dolazi kada poslednja ulazi osoba 6. Rekurzivnim pozivom dobićemo rezultat da se osobe sa težinama 3 i 4 mogu prevesti u jednoj vožnji čija je masa 7. Tada osoba 6 ne staje u taj lift i za nju ćemo morati pokrenuti novu vožnju. U tom slučaju ćemo imati takođe dve vožnje, ali će masa u poslednjoj vožnji biti 6 (što je bolje od 7). Obratimo pažnju na to da smo mi poslednju osobu fiksirali. Naime broj 7 nije globalno optimalna vrednost, ali jeste najmanja vrednost poslednje vožnje pod pretpostavkom da se osoba 3 vozila u poslednjoj vožnji.

Dakle, mi za svaki izbor osobe i iz skupa od n osoba možemo jednostavno odrediti najmanji mogući broj vožnji potrebnih da se tih n osoba preveze i u tom broju vožnji najmanju moguću masu poslednje vožnje pod uslovom da u njoj učestvuje osoba i.

Zaista, ako je za skup od $n-1$ osobe koji ne uključuje osobu mase m_i bilo potrebno k vožnji i ako je najmanja moguća masa vožnje bila u poslednjoj vožnji i iznosila m, tada ako je $m + m_i$ manje ili jednako od nosivosti, onda je za prevoz svih osoba potrebno takođe k vožnji i najmanja moguća masa poslednje vožnje ako u njoj učestvuje osoba i je $m + m_i$. Zaista, ako bi bilo moguće organizovati se drugačije, tako da se u poslednjoj vožnji osoba i vozi sa nekim drugim osobama, čija je osoba $m' < m$, tada bi se izbacivanjem osobe i dobilo da se skup od $n-1$ osobe može prevesti u k vožnji tako da je u poslednjoj vožnji masa $m' < m$, što je u suportnosti sa induktivnom hipotezom koja nam garantuje da je m najmanja moguća masa u poslednjoj vožnji ako se $n-1$ osoba prevozi u k vožnji. Ako je $m + m_i$ veće od nosivosti lifta, onda je jasno da je za prevoz svih n osoba potrebna $k+1$ vožnja i sasvim je jasno da je najmanja moguća masa poslednje vožnje u kojoj učestvuje osoba i jednaka m_i (čim je osoba i u liftu, masa ne može biti manja od m_i).

Izračunate vrednosti su nam sasvim dovoljne i da nađemo globalni optimum. Naime, u globalno optimalnoj vožnji neka se osoba mora voziti poslednja. Prilikom analize situacije u kojoj ta osoba ulazi u lift poslednja umećemo da izračunamo koliki je minimalni broj vožnji i kolika je minimalna masa u poslednjoj vožnji u kojoj ta osoba učestvuje. Međutim, to će upravo biti minimalna moguća masa u poslednjoj vožnji od svih situacija u kojima se n osoba prevozi u optimalnom broju vožnji.

Pošto se u opisanoj induktivno-rekurzivnoj konstrukciji isti rekurzivni pozivi vrše više puta, efikasnost možemo popraviti dinamičkim programiranjem. Argument svakog rekurzivnog je neki podskup polaznog skupa osoba. Postavlja se pitanje kako da znamo da smo neki takav podskup već ranije obrađivali tj. kako da

u programu predstavljamo takve podskupove. Najjednostavniji način je da koristimo kodiranje podskupova pomoću neoznačenih brojeva - bit na poziciji i će biti 1 ako i samo ako se osoba i nalazi u podskupu. Tada se u dinamičkom programiranju za pamćenje rezultata može koristiti običan niz koji je indeksiran kodovima podskupa. Još jedan trik koji malo olakšava implementaciju je da iz rekurzije izađemo još jedan korak kasnije i da za skup od 0 osoba kažemo da je minimalan broj vožnji 1 (ne 0) i da je najmanja masa u poslednjoj vožnji jednaka 0.

```
// funkcija vraća najmanji mogući broj vožnji i najmanju masu
// u poslednjoj vožnji (ujedno i najmanju masu od svih vožnji)
// ako se prevoze osobe kodirane datim podskupom
pair<int, int> brojVoznji(const vector<int>& tezine, int nosivost,
                          unsigned podskup,
                          vector<pair<int, int>>& memo) {
  // ukupan broj osoba
  int n = tezine.size();

  // proveravamo da li smo već ranije računali rešenje
  // za ovaj podskup
  if (memo[podskup].first != 0)
    return memo[podskup];

  // bazni slučaj je kada je podskup prazan
  if (podskup == 0)
    // optimalna je jedna vožnja i masa u poslednjoj vožnji je 0
    return {1, 0};
  else {
    // inicijalizacija maksimuma na "beskonačno"
    // sigurni smo da će rešenje biti manje ili jednako od ovoga
    pair<int, int> ret = {n, nosivost};
    // analiziramo sve mogućnosti za poslednju osobu koja ulazi
    for (int i = 0; i < n; i++) {
      // preskačemo osobe koje nisu u podskupu
      if (((1 << i) & podskup) != 0) {
        // rekurzivno određujemo rešenje za podskup bez osobe i
        pair<int, int> minI =
          brojVoznji(tezine, nosivost, podskup ^ (1 << i), memo);
        if (minI.second + tezine[i] <= nosivost) {
          // osoba i staje u poslednju vožnju
          // najmanja težina poslednje vožnje u osobe i jednaka
          // je minI.second + tezine[i]
          minI.second += tezine[i];
        } else {
          // osoba i ne staje u poslednju vožnju
          // povećavamo potreban broj vožnji
          minI.first++;
          // minimalna težina kada je osoba i u poslednjoj vožnji
          minI.second = tezine[i];
```

```
      }
      // ažuriramo globalni minimum
      ret = min(ret, minI);
    }
  }

  // memoizujemo i vraćamo poslednji rezultat
  return memo[podskup] = ret;
 }
}

int brojVoznji(const vector<int>& tezine, int nosivost) {
  // ukupan broj osoba
  int n = tezine.size();
  // krećemo analizu od podskupa u kome su sve osobe uključene
  unsigned podskup = (1 << n) - 1;
  // pošto je broj vožnji uvek bar 1, vrednost 0 u tablici
  // memoizacije će označavati da podskup nije ranije obrađivan
  vector<pair<int, int>> memo(1<<n, {0, 0});
  // računamo minimalni broj vožnji za pun skup
  return brojVoznji(tezine, nosivost, podskup, memo).first;
}
```

Pošto se prilikom izbacivanjem svakog elementa iz podskupa dobija neki podskup čiji je binarni kôd manji od binarnog koda polaznog podskupa, možemo se osloboditi rekurzije i primeniti dinamičko programiranje naviše i rezultate računati u rastućem redosledu binarnih kodova.

```
int brojVoznji(const vector<int>& tezine, int nosivost) {
  // ukupan broj osoba
  int n = tezine.size();
  // rezultati za sve podskupove
  vector<pair<int, int>> dp(1 << n);
  // rezultat za prazan podskup
  // potrebna je jedna vožnja i masa 0 u poslednjoj vožnji
  dp[0] = {1, 0};
  // obrađujemo neprazne podskupove u rastućem redosledu
  // binarnih kodova
  for (unsigned podskup = 1; podskup < (1<<n); podskup++) {
    // minimum inicijalizujemo na "beskonačno"
    // sigurni smo da će rešenje biti bolje od ovoga
    dp[podskup] = {n, nosivost};
    for (int i = 0; i < n; i++) {
      // preskačemo osobe koje nisu u podskupu
      if (podskup & (1 << i)) {
        // rešenje za podskup bez osobe i
        auto minI = dp[podskup ^ (1 << i)];
        if (p.second + tezine[i] <= nosivost)
          // osoba i staje u poslednju vožnju
          // najmanja težina poslednje vožnje u osobe i jednaka
```

```
          // je minI.second + tezine[i]
          minI.second += tezine[i];
        else {
          // osoba i ne staje u poslednju vožnju
          // povećavamo potreban broj vožnji
          minI.first++;
          // minimalna težina kada je osoba i u poslednjoj vožnji
          minI.second = tezine[i];
        }
        dp[podskup] = min(dp[podskup], minI);
      }
    }
  }
  // rezultat za skup u kome je svih n osoba
  return dp[(1<<n)-1].first;
}
```

Memorijska složenost ovog rešenja odgovara broju podskupova, a to je $O(2^n)$. Vremenska složenost je $O(n2^n)$, što je ogromno, ali je i dalje dosta bolje od rešenja grubom silom koje ima složenost $O(n!)$. Npr $20! \approx 2 \cdot 10^{18}$, dok je $20 \cdot 2^{20} \approx 2 \cdot 10^7$.