AiSP odgovori na pitanja

Mina Milošević

2018/2019

Contents

1 Korektnost algoritama

1.1 Algoritmi i strukture podataka - pojam, način opisivanja, svojstva

Niklaus Wirth (tvorac Pascal-a): algoritmi + strukture podataka = programi.
Algoritmi su sredstva za rešavanje precizno definisanih računskih problema.
Neformalno: algoritam je precizno definisana procedura izračunavanja tj. niz koraka izračunavanja, koja, polazeći od neke vrednosti (ili niza vrednosti, skupa vrednosti, itd.) kao ulaza proizvodi neku vrednost (ili niz vrednosti, skup vrednosti itd.) kao izlaz.
Strukture podataka predstavljaju načine organizacije i skladištenja podataka koji omogućavaju efikasan pristup željenim podacima i njihovu efikasnu modifikaciju. Strukture podataka sačinjavaju kolekcije vrednosti podataka, veze izmedju njih i funkcije i operacije koje se mogu primeniti nad tim podacima.
Načini opisa algoritama: programski kôd, pseudo-kôd, blok dijagrami toka, scratch/blockly dijagrami... Postoje precizne definicije pojma algoritma: URM mašine , Tjuringove mašine, Gedelove rekurzivne funkcije, Čerčov lambda račun...

1.2 Veza matematičke indukcije i rekurzije

Osnovni pristup konstrukciji algoritama je induktivni/rekurzivni pristup. On podrazumeva da se rešenje problema veće dimenzije pronalazi tako što umemo da rešimo problem istog oblika, ali manje dimenzije i da od rešenja tog problema dobijemo rešenje problema veće dimenzije. Pritom za početne dimenzije problema rešenje moramo da izračunavamo direktno, bez daljeg svodjenja na probleme manje dimenzije. Ako se prilikom svodjenja dimenzija problema uvek smanjuje, konstruisani algoritmi će se uvek zaustavljati. Implementacija ovako dizajniranih algoritama može, ali ne mora biti isključivo rekurzivna. Indukciju koristimo kao tehniku dokazivanja, a rekurziju kao tehniku konstrukcije algoritama.

1.3 Dokaz korektnosti rekurzivne funkcije - primer minimuma niza

Problem: Definisati funkciju koja odredjuje minimum nepraznog niza brojeva i dokaži njenu korektnost.

```
int min(int a[], int n){
    if (n == 1)
       return a[0];
    else {
      int m = min(a, n-1);
      return min2(m, a[n-1]); // min2: return a < b ? a : b;
    }
}
```

Teorema: Za svaki neprazan niz a i za svako $1 \leq$ n $\leq$ |a| poziv min(a, n) vraća najmanji među prvih n elementa niza a.

Dokaz:

Baza: n = 1, tj. poziv min(a, 1). Na osnovu definicije funkcije *min* rezultat je a[0] tj. prvi član niza a_0 i tada tvrđenje trivijalno važi.
IH: ako važi $1 \leq n-1 \leq$ |a|, tada poziv min(a, n-1) vraća najmanji od prvih $n-1$ elemenata niza a. Potrebno je da dokažemo da za n koje zadovoljava 1 < n < |a|, poziv min(a, n) vraća najmanji od prvih n elemenata niza a. Na osnovu definicije funkcije min, poziv min(a, n) će vratiti manji od brojeva m koji je rezultat poziva min(a, n-1) i a_{n-1}. Na osnovu induktivne hipoteze znamo da će m biti najmanji među prvih $n-1$ elemenata niza a. Zato će manji od broja m i n-tog elementa niza biti najmanji među prvih n elemenata niza a.

1.4 Dokaz korektnosti rekurzivne funkcije - Euklidov algoritam

Problem: Definisati funkciju koja izračunava najveći zajednički delilac dva neoznačena broja i dokaži njenu korektnost.
Izračunavanje najvećeg zajedničkog delioca dva nenegativna broja a i b, može se izvršiti korišćenjem Euklidovog algoritma. Označimo sa $a\text{div}b$ i $a\text{mod}b$ redom celobrojni količnik i ostatak pri deljenju brojeva a i b (ako je a = b · q + r, tako da je 0 ≤ r < b, tada je $a\text{div}b$ = q i $a\text{mod}b$ = r). Algoritam razmatra dva slučaja: ako je b = 0 tada je nzd(a, 0) = a i za b > 0 važi: nzd(a, b) = nzd(b, $a\text{mod}b$). Postupak se uvek zaustavlja jer je vrednost $a\text{mod}b$ uvek nenegativna (i celobrojna) i smanjuje se, pa mora nekada doći do nule.

```
int nzd(int a, int b) {
   if (b == 0) return a;
   return nzd(b, a % b);
}
```

Teorema: Poziv nzd(a, b) vraća najveći zajednički delilac neoznačenih brojeva a i b (označimo ga sa nzd(a, b)).

Dokaz:

Baza: Važi da je nzd(a, 0) = a, jer je a najveći broj koji deli i a i 0. Pošto kada je b = 0 funkcija nzd vraća vrednost a, tvrđenje važi.
IH: poziv nzd(b, a % b) vraća najveći broj n koji deli broj b i broj $a\text{mod}b$. Dokažimo da je taj broj ujedno i nzd(a, b). Važi da je a = b · q + $a\text{mod}b$. Pošto n deli i b i $a\text{mod}b$, važi da n mora da deli i a, tako da je n sigurno delilac brojeva a i b. Dokažimo još i da je najveći. Ako neko n' deli a i b, tada na osnovu a = b · q + $a\text{mod}b$ taj broj sigurno mora da deli i $a\text{mod}b$. Međutim, pošto je n najveći zajednički delilac brojeva b i $a\text{mod}b$, tada n' deli n. Dakle, svaki delilac brojeva a i b deli n, a pošto ga deli, mora mu ujedno biti manji ili jednak, pa je nzd(a, b) = n, što je upravo povratna vrednost poziva nzd(a, b).

1.5 Invarijanta petlje - pojam i primer

Invarijanta petlje je logički uslov koji je ispunjen neposredno pre i neposredno nakon svakog izvršavanja tela petlje. Da bismo dokazali da je neki uslov invarijanta petlje indukcijom na osnovu broja izvršavanja tela petlje, dovoljno je da dokažemo: da taj uslov važi pre prvog ulaska u petlju; i da iz pretpostavke da taj uslov važi pre nekog izvršavanja tela petlje i da je uslov petlje ispunjen dokažemo da taj uslov važi i nakon izvršavanja tela petlje. Te dve činjenice nam, na osnovu induktivnog argumenta, garantuju da će uslov biti ispunjen tokom svake iteracije petlje, kao i nakon izvršavanja cele petlje (ako se ona ikada zaustavi), tj. da će biti invarijanta.

1.6 Invarijanta petlje - primer minimuma niza

```
int main() {
  int a[] = {3, 5, 4, 1, 6, 2, 7};
  int n = sizeof(a) / sizeof(int);
  int m = a[0];
  for (int i = 1; i < n; i++)
     if (a[i] < m)
        m = a[i];
  cout << m << endl;
}
```

Lema: Ako je niz a dužine n $\geq$ 1, uslov da je 1 $\leq$ i $\leq$ n i da je m minimum prvih i elemenata niza je invarijanta petlje.

Dokaz:

Pre ulaska u petlju vrši se inicijalizacija i=1, dok promenljiva m sadrži vrednost a_0 što je zaista minimum jednočlanog prefiksa niza a. Pošto pretpostavljamo da je niz neprazan važi da je i = 1 $\leq$ n. Pretpostavimo da tvrđenje važi nakon ulaska u petlju tj. da je vrednost promenljive m jednaka minimumu prvih i članova niza, kao i da je uslov petlje ispunjen tj. da je i < n. Nakon izvršavanja tela petlje nova vrednost promenljive m biće jednaka manjoj od vrednosti m i a_i. Pošto je na osnovu pretpostavke m jednak minimumu prvih i elemenata niza, m je jednak minimumu brojeva a_0,..., a_i, što je tačno minimum prvih i+1 elemenata niza. Pošto je nakon izvršavanja tela petlje (u koje po dogovoru ubrajamo i korak petlje) vrednost promenljive i uvećana za jedan, važi da je i=i+1, pa je zaista m minimum prvih i elemenata niza. Takođe, pošto je važilo da je i < n, nakon izvršavanja tela petlje, važiće i da je i $\leq$ n.

Teorema: Nakon izvršavanja petlje, promenljiva m sadrži minimum celog niza.

Dokaz:

Na osnovu invarijante je 1 $\leq$ i $\leq$ n, a pošto po završetku petlje njen uslov nije ispunjen važi da je i = n. Na osnovu invarijante promenljiva m sadrži minimum prvih i elemenata niza, a pošto je i = n, gde je n broj članova niza, to je zapravo minimum celog niza.

1.7 Invarijanta petlje - primer Euklidovog algoritma

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

Lema: nzd(a, b) = nzd(a_0, b_0), gde su a_0 i b_0 vrednosti argumenata funkcije.

Dokaz:

Pre petlje važi a = a_0 i b = b_0, pa je invarijanta trivijalno zadovoljena. Pretpostavimo da invarijanta važi na ulasku u petlju i da je uslov petlje zadovoljen tj. da je b=0. Nakon izvršavanja tela petlje, nove vrednosti promenljivih a i b su a=b i b=$a \bmod b$, pa je nzd(a , b) = nzd(b, $a \bmod b$). Znamo da je nzd(b, $a \bmod b$) = nzd(a, b), a na osnovu pretpostavke da invarijanta važi na ulazu u petlju, da je nzd(a,b) = nzd(a_0,b_0).

Teorema: Poziv *nzd(a, b)* vraća nzd(a, b).

Dokaz:

Kada se petlja završi, njen uslov nije ispunjen, pa je b = 0. Tada, pošto invarijanta važi i pošto je nzd(a, 0) = a, važi da je nzd(a_0, b_0) = a. Pošto funkcija vraća vrednost a, ona korektno izračunava NZD argumenata funkcije.

1.8 Invarijanta petlje - određivanje cifara u binarnom zapisu broja

Problem: Definisati algoritam koji određuje binarne cifre u zapisu datog neoznačenog broja n i dokaži njegovu korektnost.

```
bool binarneCifre[32] = {false};
for (int i = 0; n > 0; i++, n /= 2)
  binarneCifre[i] = n % 2;
```

Lema: Uslov $2^i \cdot$ n + b = n_0 , gde je b broj trenutno zapisan u nizu binarnih cifara, dok je n_0 početna vrednost neoznačenog broja n, je invarijanta petlje.

Dokaz:

Zaista na početku je n = n_0, i = 0 i b = 0 pa tvrđenje važi. Pretpostavimo da tvrđenje važi pri ulasku u petlju. Promenljive se tokom izvršavanja tela i koraka petlje menjaju na sledeći način: n' = $n \operatorname{div} 2$, b' = b + $2^i \cdot n \bmod 2$ i i' = i + 1. Tada je $2^{i'} \cdot$ n' + b' = $2^{i+1} \cdot (n \operatorname{div} 2)$ + b + 2^i.
$n \bmod 2 = 2^i \cdot (2 \cdot (n \operatorname{div} 2) + n \bmod 2)$ + b. Na osnovu definicije celobrojnog

deljenja važi da je $2 \cdot (n\text{div}2) + n\text{mod}2 = \text{n}$, pa je vrednost prethodnog izraza jednaka $2^i \cdot \text{n} + \text{b}$, a na osnovu pretpostavke o tome da invarijanta važi na ulasku u telo petlje znamo da je to jednako n_0.

Teorema: Po završetku niz sadrži binarni zapis neoznačenog broja n.

Dokaz:

Kako je po izlasku iz petlje $\text{n} = 0$, na osnovu invarijante važi da je $\text{b} = \text{n}_0$ tj. da niz sadrži binarni zapis polaznog broja.

1.9 Invarijanta petlje - particionisanje niza

Problem: Zadatak paricionisanja u algoritmu QuickSort je da rasporedi elemente tako da prvo idu oni koji su manji ili jednaki pivotirajućem elementu, zatim pivot, i na kraju oni koji su strogo veći od pivota. Definiši algoritam kojim se može vršiti particionisanje dela niza određenog intervalom pozicija [l, d] za $0 \leq \text{l} \leq \text{d} < \text{n}$.

```
int m = l+1, v = d;
while (m <= v) {
  if (a[m] <= a[l])
     m++;
  else if (a[v] > a[l])
     v--;
  else
     swap(a[m++], a[v--]);
}
swap(a[l], a[m-1]);
```

Dokaz:

Invarijantu možemo organizovati tako da se pivot nalazi na poziciji l, da su na pozicijama iz intervala (l, m) elementi manji ili jednaki od pivota, a da su na pozicijama iz intervala (v, d] elementi veći od pivota, pri čemu je $\text{l} < \text{m}$, $\text{v} \leq \text{d}$ i $\text{m} \leq \text{d} + 1$. Na početku možemo inicijalizovati promenljive tako da je $\text{m} = \text{l} + 1$ i $\text{v} = \text{d}$ i invarijanta će biti ispunjena. U intervalu [m, v] se nalaze elementi čiji status još nije poznat. Pokušavamo u svakom koraku da suzimo taj interval, sve dok se ne isprazni, tj. sve dok je $\text{m} \leq \text{v}$. Ako je a_m manji ili jednak pivotu sve što treba da uradimo je da povećamo m za jedan. Ako je a_v veći od pivota, sve što treba da uradimo je da smanjimo v za jedan. Ako nijedno od ta dva nije ispunjeno, tada možemo da zamenimo a_m i a_v, da povećamo m za jedan i smanjimo v za jedan i invarijanta će ostati da važi. Kada se petlja završi, poslednji element koji je manji ili jednak pivotu se nalazi na poziciji m - 1, pa ga razmenjujemo sa pivotom.

1.10 Invarijanta petlje - trobojka

Problem: Definisati algoritam koji efikasno reorganizuje elemente niza brojeva tako da prvo idu svi njegovi negativni članovi, zatim sve nule i na kraju svi pozitivni članovi. Redosled unutar grupe negativnih i unutar grupe pozitivnih članova nije bitan.

```
int i = 0, N = 0, P = n;
while (i < P) {
    if (a[i] > 0)
     swap(a[--P], a[i]);
    else if (a[i] < 0)
     swap(a[N++], a[i++]);
    else
     i++;
}
```

Lema: Invarijanta prethodne petlje je da je multiskup elemenata niza u svakom koraku petlje isti multiskupu elemenata polaznog niza, kao i to da se u intervalu [0, N) nalaze negativni brojevi, u intervalu [N, i) se nalaze nule, dok se u intervalu [P, n) nalaze pozitivni brojevi. Pri tom važi da je: $0 \leq N \leq i \leq P \leq n$

Dokaz:

Pre ulaska u petlju, nakon inicijalizacije N = 0, i = 0, P = n invarijanta je trivijalno zadovoljena (jer su sva tri intervala prazna). Pretpostavimo da invarijanta važi prilikom ulaska u telo petlje, i dokažimo da će ona ostati da važi i nakon izvršavanja tela petlje. Razlikujemo nekoliko slučajeva. Ako je a_i negativan broj, menjamo ga sa elementom a_N i uvećavamo i i i N . Ako je i = N, zamena ne menja niz. Pošto je na osnovu invarijante poznato da su u intervalu [0, N) bili svi negativni elementi, a na poziciji i = n je negativan broj, uvećanje N za jedan održava taj uslov. Nakon uvećanja obe promenljive interval [N, i) ostaje prazan, pa su svi njegovi elementi (kojih nema) nule. Pošto se P ne menja, u intervalu [P, n) su i dalje svi pozitivni brojevi. Ako je N < i, onda na osnovu invarijante znamo da je a_N nula. Razmenom sa a_i dobijamo situaciju da je na polju a_N sada negativan broj, a da je na polju a_i sada nula. Nakon uvećanja N' = N + 1, i' = i + 1 možemo da tvrdimo da su intervalu [0, N) sve negativni brojevi. Takođe, možemo da tvrdimo da su u intervalu [N' , i') sve nule. Deo niza u intervalu [P, n) nije menjan (jer je N<i<P), pa su u njemu i dalje svi pozitivni elementi. Nakon uvećanja N i i, međusobni raspored promenljivih i dalje ostaje da važi. Ako je a_i pozitivan broj, menjamo ga sa a_{P-1} i umanjujemo P. Pošto je $N \leq i < P$, intervali [0, N) i [N, i) ostaju nepromenjeni, pa invarijanta za njih i dalje važi. Na osnovu invarijante znamo da su u intervalu [P, n) svi elementi pozitivni, a pošto je nakon razmene pozitivan i a_{P-1} , onda su pozitivni i svi elementi u intervalu [P' , n), za umanjenu vrednost P' = P - 1. U ovom slučaju ne znamo kakav je status elementa a_{P-1} pre razmene, pa

kada njega dovedemo na mesto i, promenljivu i ne menjamo, da bi se njegov status ispitao u sledećem koraku. Na kraju, ako je $a_i = 0$, samo povećavamo i i invarijanta ostaje očuvana. Elementi u intervalima [0, N) i [P, n) nisu menjani, pa ostaju negativni tj. pozitivni. Na osnovu invarijante znamo da su svi elementi u intervalu [N, i) nule, pa pošto je nula i a_i , nule su svi elementi intervala pozicija [N, i') za uvećano i' = i + 1.

Teorema: Ako je n dužina niza a, poziv trobojka(a, n) reorganizuje niz tako da idu prvo negativni elementi, zatim nule i na kraju pozitivni elementi.

Dokaz:

Kada se petlja završi, njen uslov nije ispunjen tj. ne važi da je i<P, pa pošto je na osnovu invarijante i $\leq$ P , važi da je i = P. Zato su na osnovu invarijante elementi a_0,..., a_{N-1} negativni, elementi a_N,..., a_{P-1} nule, a elementi a_P,..., a_{n-1} su pozitivni.

1.11 Dokaz korektnosti binarne pretrage prelomne tačke

Problem: U nizu se nalaze prvo neparni, pa zatim parni celi brojevi. Definisati funkciju koja pronalazi poziciju prvog parnog broja (ako takvog elementa nema, funkcija vraća dužinu niza).

```
int prviParan(int a[], int l, int d) { // rekurzivno
  if (l > d)
     return l;
  int s = l + (d - l) / 2;
  if (a[s] % 2 == 0)
     return prviParan(a, l, s-1);
  else
     return prviParan(a, s+1, d);
}
```

Lema: Za $0 \leq l \leq d + 1 \leq n$ poziv *prviParan(a, l, d)* vraća poziciju prvog parnog elementa u nizu a_l,..., a_d, tj. vrednost d + 1 ako su svi elementi neparni.

Dokaz:

Baza: l > d. Na osnovu pretpostavke tada važi da je l = d + 1. Pošto je niz a_l,..., a_d prazan u njemu nema neparnih elemenata, tako da funkcija vraća l = d + 1, što je tražena vrednost.

Pretpostavimo da rekurzivni pozivi vraćaju ispravne pozicije i dokažimo da poziv *prviParan(a, l, d)* vraća ispravnu poziciju i kada nije l > d, tj. kada se vrše rekurzivni pozivi. Tada je $0 \leq l \leq d$. Neka je s = l+[(d–l)/2]. Važi da je $l \leq s \leq d$. Ako je a_s paran, onda prvi paran element može biti samo neki od elemenata a_l,..., a_s. Pošto je $0 \leq l \leq (s\ \ 1) + 1$, na osnovu induktivne hipoteze rekurzivni poziv *prviParan(a, l, s-1)* vraća poziciju prvog parnog među elementima a_l,..., a_{s-1} ili broj s ako su svi neparni, što je tačno pozicija koju tražimo. Ako je a_s neparan, tada na osnovu pretpostavke zadatka o rasporedu elemenata u nizu a znamo da su neparni i svi elementi u nizu a_l,..., a_s. Dakle prvi paran element se

može javiti samo u nizu a_{s+1},..., a_d. Pošto je $0 \leq l$ i $l \leq s \leq d$ važi da je $0 < s + 1 \leq d + 1$, tako da na osnovu induktivne hipoteze važi da rekurzivni poziv p*rviParan(a, s+1, d)* vraća poziciju prvog parnog elementa u nizu a_{s+1},..., a_d ili broj $d + 1$ ako takvog elementa nema. Pošto su a_l,..., a_s svi neparni, vrednost rekurzivnog poziva je upravo tražena pozicija.

Teorema: Ako je n dužina niza a i u nizu se nalaze prvo neparni, pa zatim parni elementi, poziv *prviParan(a, n)* vraća poziciju prvog parnog elementa u nizu ili n ako takav element ne postoji.

Dokaz:

Pošto je $n \geq 0$, važi i da je $0 \leq 0 \leq (n\ \ 1) + 1$. Na osnovu dokazane leme poziv *prviParan(a, 0, n-1)* vraća traženu poziciju (jer je dužina niza n).

```
int prviParan(int a[], int n) { // iterativno
  int l = 0, d = n-1;
  while (l <= d) {
     int s = l + (d - l) / 2;
     if (a[s] % 2 == 0)
        d = s-1;
     else
        l = s+1;
  }
  return l;
}
```

Lema: Važi da je $0 \leq l \leq d+1$, svi elementi u nizu *a* na nenegativnim pozicijama strogo manjim od l su neparni, a svi elementi u nizu *a* na pozicijama strogo većim od d i strogo manjim od n su parni.

Dokaz:

Invarijanta važi pre petlje (iz l=0, d=n-1 i $n \geq 0$ sledi da je $0 \leq l \leq d + 1$). Pretpostavimo da invarijanta važi pre ulaska u petlju i dokažimo da važi i nakon izvršavanja tela petlje. Neka je s = l + [(d-l) / 2]. Pošto je $l \leq d$ važi i da je $l \leq s \leq d$. Ako je a_s paran, tada su na osnovu datog uslova parni i svi elementi iza pozicije s, zaključno sa pozicijom n - 1. Postavljanjem d na vrednost s - 1 invarijanta ostaje održana jer su zaista svi elementi na pozicijama strogo većim od d = s - 1, a strogo manjim od n parni. Ako je a_s neparan, tada su na osnovu uslova zadatka neparni i svi elementi na pozicijama od 0 do s. Postavljanjem l na vrednost s + 1 invarijanta ostaje održana jer su zaista svi elementi na nenegativnim pozicijama strogo manjim od l = s + 1 neparni.

Teorema: Ako je n dužina niza *a* i u nizu se nalaze prvo neparni, pa zatim parni elementi, poziv *prviParan(a, n)* vraća poziciju prvog parnog elementa u nizu ili n ako takav element ne postoji.

Dokaz:

Kada se petlja završi na osnovu invarijante važi da je $0 \leq l \leq d + 1$ i pošto uslov održanja petlje $l \leq d$ nije ispunjen mora da važi da je l=d+1.

Ako je $l < n$, tada a_l mora biti prvi paran broj u nizu. Na osnovu invarijante svi elementi na nenegativnim pozicijama strogo manjim od l su neparni, a $l = d + 1$ je pozicija koja je strogo veća od d a strogo manja od n, pa se na njoj, opet na osnovu invarijante, mora nalaziti paran broj. Ako je l=n, na osnovu invarijante znamo da su svi elementi na nenegativnim pozicijama strogo manjim od $l = n$ neparni, pa u nizu nema parnih elemenata. Funkcija vraća $l = n$, što je dužina niza, pa i u ovom slučaju funkcija vraća korektnu vrednost.

2 Složenost aloritama

2.1 Vrste složenosti i načini njihove analize. Asimptotska notacija

Pored svojstva ispravnosti programa, veoma je važno i pitanje koliko program zahteva vremena i memorije za svoje izvršavanje. Često nije dovoljno imati informaciju o tome koliko se neki program izvršava za neke konkretne ulazne vrednosti, već je potrebno imati neku opštiju procenu za proizvoljne ulazne vrednosti. Obično se razmatraju: vremenska i prostorna složenost algoritma. Mogu se razmatrati u terminima konkretnog vremena/prostora utrošenog za neku konkretnu ulaznu veličinu ili u terminima asimptotskog ponašanja vremena/prostora kada veličina ulaza raste.
Vreme izvršavanja programa može biti procenjeno za neke konkretne veličine ulazne vrednosti i neko konkretno izvršavanje. Veličina ulazne vrednosti može biti broj ulaznih elemenata koje treba obraditi, sam ulazni broj koji treba obraditi, broj bitova potrebnih za zapisivanje ulaza koji treba obraditi, itd. Uvek je potrebno eksplicitno navesti u odnosu na koju veličinu se razmatra složenost.
Često se algoritmi ne izvršavaju isto za sve ulaze istih veličina, pa je potrebno naći način za opisivanje i poređenje efikasnosti različitih algoritama:

• *Analiza najgoreg slučaja* zasniva procenu složenosti algoritma na najgorem slučaju. Ta procena može da bude varljiva, ali predstavlja dobar opšti način za poređenje efikasnosti algoritama.

• U nekim situacijama moguće je izvršiti *analizu prosečnog slučaja* i izračunati prosečno vreme izvršavanja algoritma, ali da bi se to uradilo, potrebno je precizno poznavati prostor dopuštenih ulaznih vrednosti i verovatnoću da se svaka dopuštena ulazna vrednost pojavi na ulazu programa. U slučajevima kada je bitna garancija efikasnosti svakog pojedinačnog izvršavanja programa procena prosečnog slučaja može biti varljiva.

Nekada se analiza vrši tako da se proceni ukupno vreme potrebno da se izvrši određen broj srodnih operacija. Taj oblik analize naziva se *amortizovana analiza* i u tim situacijama nam nije bitna raspodela vremena na pojedinačne operacije, već samo zbirno vreme izvršavanja svih operacija. Stvarno vreme izvršavanja zavisi i od konstanti sakrivenih u asimptotskim oznakama, međutim, asimptotsko ponašanje obično prilično dobro određuje njegov red veličine.
Algoritmi čija je složenost odozgo ograničena polinomskim funkcijama smatraju se efikasnim. Algoritmi čija je složenost ograničena odozdo eksponencijalnom ili faktorijelskom funkcijom se smatraju neefikasnim. Gornja granica složenosti se obično izražava korišćenjem O-notacije.
Definicija: Ako postoje pozitivna realna konstanta c i prirodan broj n_0 takvi da za funkcije f i g nad prirodnim brojevima važi $f(n) \leq c \cdot g(n)$ za sve prirodne brojeve n veće od n_0 onda pišemo $f(n) = O(g(n))$ i čitamo „f je veliko ‚o‘ od g“.
Definicija: Ako postoje pozitivne realne konstante c1 i c2 i prirodan broj n_0 takvi da za funkcije f i g nad prirodnim brojevima važi $c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$ za sve prirodne brojeve n veće od n_0 , onda pišemo $f(n) = \theta(g(n))$ i čitamo „f e veliko ‘teta’ od g“.

Navedimo karakteristike ovih osnovnih klasa složenosti:

- O(log$n$) - izuzetno efikasno, npr. binarna pretraga;
- O($\sqrt{n}$) - npr. ispitivanje da li je broj prost, faktorizacija;
- O(n) - optimalno, kada je za rešenje potrebno pogledati ceo ulaz, npr. minimum/maksimum;
- O($n$log$n$) - algoritmi zasnovani na dekompoziciji, sortiranju, korišćenju struktura podataka sa logaritamskim vremenom pristupa, npr. sortiranje objedinjavanjem;
- O(n^2) - ugnežđene petlje, npr. sortiranje selekcijom;
- O(n^3) višestruko ugnežđene petlje, npr. množenje matrica;
- O(2^n) - ispitivanje svih podskupova;
- O($n!$) - ispitivanje svih permutacija.

2.2 Rekurentne jednačine - primeri

Složenost rekurzivnih funkcija se često može opisati rekurentnim jednačinama. Rešenje rekurentne jednačine je funkcija *T(n)* i za rešenje ćemo reći da je u zatvorenom obliku ako je izraženo kao elementarna funkcija po n (i ne uključuje sa desne strane ponovno referisanje na funkciju T). Često ćemo se zadovoljiti da umesto potpuno preciznog rešenja znamo samo njegovo asimptotsko ponašanje. U prvoj grupi se problem svodi na problem dimenzije koja je tačno za jedan manja od dimenzije polaznog problema:

- T(n) = T(n - 1) + O(1), T(0) = O(1), O(n)
- T(n) = T(n - 1) + O(log$n$), T(0) = O(1), O($n$log$n$)
- T(n) = T(n - 1) + O(n), T(0) = O(1), O(n^2)

U drugoj grupi se problem svodi na dva (ili više) problema čija je dimenzija za jedan ili dva manja od dimenzije polaznog problema. To obično dovodi do eksponencijalne složenosti.

- T(n) = 2T(n - 1) + O(1), T(0) = O(1), O(2^n)
- T(n) = T(n - 1) + T(n - 2) + O(1), T(0) = 1, O(2^n)

U narednoj grupi se problem svodi na jedan (ili više) potproblema koji su značajno manje dimenzije od polaznog (obično su bar duplo manji). Ovo dovodi do polinomijalne složenosti, pa često i do veoma efikasnih rešenja.

- T(n) = T(n/2)+O(1), T(0) = O(1), O(log$n$)
- T(n) = T(n/2) + O(n), T(0) = O(1), O(n)
- T(n) = 2T(n/2) + O(1), T(0) = O(1), O(n)
- T(n) = 2T(n/2) + O(n), T(0) = O(1), O($n$log$n$)

2.3 Master teorema

Teorema: Rešenje rekurentne relacije *T(n) = aT(n/b) + cn^k*, gde su a i b celobrojne konstante takve da važi a $\geq$ 1 i b $\geq$ 1, i c i k su pozitivne realne konstante je: T(n) = Θ(n$^{log_b a}$), log$_b$a > k
= Θ(n^{k}log$n$), log$_b$a = k
= Θ(n^k), log$_b$a < k

U *prvom slučaju* se dobija drvo rekurzivnih poziva čiji broj čvorova dominira poslom koji se radi u svakom čvoru. Razmotrimo jednačinu T(n) = 2·T(n/2) + O(1), T(1) = O(1). Drvo će sadržati O(n) čvorova, a u svakom čvoru će se vršiti posao koji zahteva O(1) operacija. Odmotavanjem rekurentne jednačine, dobijamo T(n) = 2·T(n/2) + O(1) = 4·T(n/4) + 2·O(1) + O(1) = 8·T(n/8) + 4·O(1) + 2·O(1) + O(1) = 2^k·T(n/2^k) + (2^{k-1} + ... + 2 + 1) · O(1). Ako je n = 2^k dobijamo da je n/2^k = 1, pa pošto je na osnovu formule za zbir geometrijskog niza 2^{k-1} + ... + 2 + 1 = 2^k 1, složenost je Θ(n). I kada n nije stepen dvojke, dobija se isto asimptotsko ponašanje.

U *drugom slučaju* su broj čvorova i posao koji se radi uravnoteženi. Razmotrimo jednačinu T(n) = 2·T(n/2)+c·n, T(1) = O(1) i ponovo pokušajmo da je odmotamo. T(n) = 2·T(n/2) + c·n = 2·(2·T(n/4) + c·n/2) + c·n = 4T(n/4) + c·n + c·n = 4(2T(n/8) + c·n/4) + 2·c·n = 8T(n/8) + 3·c·n =...= 2^k·T(n/2^k) + k·c·n. Ako je n = 2^k posle k = $\log_2 n$ koraka n/2^k će dostići vrednost 1 tako da će zbir biti reda veličine n·O(1) + $\log_2 n$ · c · n = Θ($n \log n$).

U *trećem slučaju* posao koji se radi u čvorovima dominira brojem čvorova. Razmotrimo jednačinu T(n) = T(n/2)+cn, T(1) = O(1). Njenim odmotavanjem dobijamo da je T(n) = T(n/2) + cn = T(n/4) + cn/2 + cn = T(n/8) + cn/4 + cn/2 + cn =...= T(n/2^k) + cn(1/2^{k-1} +...+ 1/2 + 1). Ponovo, ako je n = 2^k , tada je prvi član jednak O(1) i pošto je na osnovu formule za zbir geometrijskog niza 1/2^{k-1} + . . . + 1/2 + 1 = (1 - (1/2)^k)/(1 - (1/2)) = 2 - 2/n zbir je jednak O(1) + cn(2 - 2/n) = Θ(n).

2.4 Složenost Euklidovog algoritma

Problem: Proceni složenost Euklidovog algoritma za određivanje NZD dva pozitivna prirodna broja.

```
int nzd(int a, int b) { // Zaustavljanje se obezbedjuje time sto pri
    if (a > b)          // svakom rekurzivnom pozivu jedan od
        return nzd(a-b, b); // elemenata opada. Do kraja se stize u
    else if (b > a)         // najvise a+b koraka. Najgori slucaj
        return nzd(a, b-a)  // nastupa ako je jedan od dva broja
    else                    // jednak 1, a drugi dosta veci od njega.
        return a;           // Ukupna slozenost je O(a+b).
}
```

```
int nzd(int a, int b) {
    return b == 0 ? a : nzd(b, a % b);
}
```

Posle najviše jednog koraka vazi: a > b. Posle dve iteracije se od para (a, b) dolazi do (a*mod*b, b*mod*(a*mod*b)). Dokažimo: a*mod*b < a/2. Ako je b ≤ a/2, tada a*mod*b < b ≤ a/2. U suprotnom, za b > a/2 važi a*mod*b = a-b < a/2. Do vrednosti 1 prvi argument stize u logaritamskom broju koraka u odnosu na veći od polazna dva broja i tada drugi broj sigurno dostiže nulu, O(log(a + b)).

2.5 Eratostenovo sito - složenost i korektnost

Problem: Definisati efikasnu funkciju koja za sve brojeve manje od datog određuje da li su prosti i njenim korišćenjem izračunava koliko ima prostih brojeva manjih od datog.
Direktan nacin (slozenost je $O(n^{3/2})$):

```
void sviProsti(bool prost[], int n) {
    for (int i = 0; i <= n; i++)
        prost[i] = jeProst(i); // fja koja vraca true ili false
}
```

Bolji način je da se upotrebi Eratostenovo sito u kom se redom precrtavaju svi umnošci prostih brojeva. Postupak zahteva niz dužine n tako da i-ti element odgovara broju i. Inicijalno su svi brojevi označeni kao prosti i u toku postupka se eliminišu svi oni koji to nisu.

```
void eratosten(bool prost[], int n) {
    for (int i = 0; i <= n; i++)
        prost[i] = true;
    prost[0] = prost[1] = false;
    for (int d = 2; d*d <= n; d++)
        if (prost[d])
            for (int i = d+d; i <= n; i += d)
                prost[i] = false;
}
```

Lema: Invarijanta spoljašnje petlje je da je $2 \leq d \leq [\sqrt{n}] + 1$ i da su precrtani 0, 1 i tačno svi oni brojevi manji ili jednaki n koji imaju prave delioce u intervalu [2, d).

Dokaz:

Na ulazu u petlju je d = 2 i precrtani su samo 0 i 1, što je u skladu sa invarijantom, jer je interval [2, 2) prazan.

Pretpostavimo da tvrđenje važi na ulasku u telo spoljne petlje. Pošto pri ulasku u petlju važi $d^2 \leq n$, važi i $d \leq [\sqrt{n}]$. Onda važi i $d+1 \leq [\sqrt{n}]+1$. Pošto su na osnovu pretpostavke precrtani svi oni brojevi manji ili jednaki n koji imaju delioce iz intervala [2, d) i pošto je d'=d+1, potrebno je pokazati da su nakon izvršavanja tela petlje precrtani i svi oni brojevi manji ili jednaki n koji su deljivi brojem d. Ako u nizu *prost* piše da d nije prost, znamo da njega deli neki delilac t iz intervala [2, d). Svi brojevi manji ili jednaki od n koji imaju delioce iz intervala [2, d) su na osnovu pretpostavke već precrtani. Oni brojevi koje deli d, deli i t, pa su na osnovu pretpostavke oni već precrtani. Dakle, invarijanta se održava i kada se ništa ne uradi. Ako u nizu *prost* piše da je d prost, precrtavaju se svi brojevi koji su manji ili jednaki od n i deljivi su brojem d, osim broja d. Ovim se eksplicitno invarijanta proširuje i na njih i ostaje održana.

Teorema: Nakon završetka funkcije nisu precrtani tačno brojevi koji su prosti.

Dokaz:

Pošto se petlja završila, važi da je d > $[\sqrt{n}]$, a pošto na osnovu invarijante znamo da je d ≤ $[\sqrt{n}]$, važi da je d = $[\sqrt{n}]$ + 1. Na osnovu invarijante znamo da su precrtani 0, 1 i tačno svi oni brojevi manji ili jednaki od n koji imaju delioce iz intervala [2, $\sqrt{n}$]. Pošto iz x ≤ n sledi $\sqrt{x} \leq \sqrt{n}$, znamo da su precrtani tačno oni brojevi x koji imaju prave delioce iz intervala [2, $\sqrt{x}$], što znači da su neprecrtani ostali tačno prosti brojevi.

Analiza složenosti:

Procenimo broj izvršavanja tela unutrašnje petlje. U početnom koraku spoljne petlje precrtava se oko n/2 elemenata. U narednom, oko n/3. U narednom koraku je broj 4 već precrtan, pa se ne precrtava ništa. U narednom se precrtava oko n/5, nakon toga opet ništa, zatim n/7 itd. U poslednjem koraku se precrtava oko n/$\sqrt{n}$ elemenata. Broj precrtavanja je jednak:

$$n/2 + n/3 + n/5 + ... + n/\sqrt{n} = n \cdot \sum_{dprost} 1/d$$

Zbir H(m) = 1 + 1/2 + 1/3 +...+ 1/m je asimptotski jednak log$m$, pa znamo da taj zbir divergira. Kada se sabiranje vrši samo po prostim brojevima, tada se zbir ponaša kao logaritam harmonijskog zbira, tj. kao loglog$m$. Dakle, u našem primeru možemo zaključiti da je broj precrtavanja jednak n·loglog$\sqrt{n}$. Pošto je loglog$\sqrt{n}$ = loglogn$^{1/2}$ = log(1/2logn) = log*1/2* + loglog$n$, važi da je složenost Eratostenovog sita *O(n·loglogn)*. Funkcija *loglogn* sporo raste, pa se za sve praktične potrebe Eratostanovo sito može smatrati linearnim u odnosu na n.

2.6 Složenost particionisanja

Problem: Proceniti složenost narednog algoritma particionisanja niza dužine n ≥ 1.

```
int m = 1, v = n-1;
while (m <= v) {
    while (m <= v && a[m] <= a[0])
        m++;
    while (m <= v && a[v] > a[0])
        v--;
    if (m < v)
        swap(a[m], a[v]);
}
swap(a[0], a[m-1]);
```

Koriste se dva pokazivača m i v koja praktično kreću sa dva kraja niza i približavaju se jedan drugome, sve dok se ne susretnu. Najveći broj koraka koji je moguće napraviti je ograničen dužinom niza tj. ukupna složenost je *O(n)*.U svakom koraku petlje pomera se bar jedan od pokazivača. Bez obzira da li iteracija prestaje kada pokazivači stignu do kraja niza ili dok se ne susretnu, vreme izvršavanja takvih algoritama je linearno tj. *O(n)*, gde je n dužina niza.

2.7 Prosečna složenost algoritma QuickSort

Upečatljivo svojstvo algoritma QuickSort je efikasnost u praksi nasuprot kvadratnoj složenosti najgoreg slučaja. Ovo zapažanje sugeriše da su najgori slučajevi retki i da je prosečna složenost ovog algoritma osetno povoljnija.
Pretpostavimo da se svaki element sa jednakom verovatnoćom može izabrati za pivot i da je jednaka verovatnoća da pivot nakon particionisanja završi na bilo kojoj poziciji od 0 do n - 1. Ako brojimo samo upoređivanja (broj zamena je manji ili jednak broju upoređivanja), složenost particionisanja je n - 1. Tada prosečna složenost zadovoljava narednu rekurentnu jednačinu:

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1))$$

Prvi sabirak se kreće od T(0) do T(n - 1), a drugi od T(n - 1) do T(0), tako da se svako *T(i)* javlja tačno dva puta. Zato za n $\geq$ 1 važi:

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

Ovo je jednačina sa *potpunom istorijom* (vrednost *T(n)* se izračunava preko svih prethodnih vrednosti *T(i)*). Jedan način da se istorija eliminiše je da se posmatraju razlike između susednih članova niza čime se dobija jednačina koja opisuje vezu između dva susedna člana. Svaki od sabiraka *T(i)* se deli sa *n*, te pre oduzimanja svaki od uzastopnih članova treba pomnožiti sa odgovarajućim faktorom. Tako se dobija:

$$nT(n) - (n-1)T(n-1) = (n(n-1) + 2\sum_{i=0}^{n-1} T(i)) - ((n-1)(n-2) + 2\sum_{i=0}^{n-2} T(i))$$

$$= 2(n-1) + 2T(n-1)$$

$$T(n) = \frac{2(n-1)}{n} + \frac{n+1}{n}T(n-1), \quad \frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

Korišćenjem činjenice da je T(0) = 0 dobijamo:

$$\frac{T(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{2(n-2)}{(n-1)n} + ... + \frac{2(1-1)}{1(1+1)} = 2\sum_{i=1}^{n}\frac{i-1}{i(i+1)}$$

Kako izračunati zbir? Tome pomaže razdvajanje sabiraka na parcijalne razlomke. Iz jednačine sledi da je A_i + A + B_i = *i* - 1, pa je A=-1, B=2. Zato je:

$$\sum_{i=1}^{n}\frac{i-1}{i(i+1)} = \frac{1}{2} + ... + \frac{1}{n} - 1 + \frac{2}{n+1},$$

pa je:

$$T(n) = 2(n+1)(1 + \frac{1}{2} + ... + \frac{1}{n}) - 4n = \Theta(2(n+1)(logn+\gamma) - 4n) = \Theta(nlogn).$$

2.8 Amortizovana analiza složenosti - primer dinamičkog niza

U nekim situacijama se izvesne operacije ponavljaju puno puta tokom izvršavanja programa. U mnogim situacijama možemo da dopustimo da je pojedinačno izvršavanje neke operacije traje i malo duže, ako smo sigurni da više izvršavanja te operacije u zbiru neće trajati predugo. Analiza ukupne dužine trajanja većeg broja operacija naziva se *amortizovana analiza složenosti*.
Amortizovana cena izvršavanja n operacija podrazumeva količnik njihove ukupne dužine izvršavanja i broja n.
Primer: Jedna od najčešće upotrebljavanih struktura podataka je dinamički niz. Koliko je potrebno vremena da se u njega smesti n elemenata?
Kada u nizu nema dovoljno prostora da se smesti naredni element, niz se dinamički realocira. U najgorem slučaju ovo podrazumeva kopiranje starog sadržaja niza na novu lokaciju, što je operacija složenosti *O(m)*, gde je m broj elemenata upisanih u niz. Postavlja se pitanje kako prilikom realokacija određivati broj elemenata proširenog niza.
Jedna strategija može biti aritmetička i ona podrazumeva da se krenuvši od nekog inicijalnog kapaciteta niza 0 prilikom svake realokacije veličina niza poveća za neki broj k. Izbrojmo koliko je puta potrebno izvršiti upis elementa u niz, pri čemu tu računamo upise novih elemenata i upise nastale tokom pomeranja postojećih elemenata tokom realokacije. U prvom koraku primene aritmetičke strategije alociramo k elemenata i zatim u k narednih koraka vršimo upis po jednog elementa. Onda vršimo realokaciju na veličinu *2k* i pritom prepisujemo prvih k elemenata niza. Nakon toga upisujemo narednih k elemenata, a onda prilikom realokacije prepisujemo *2k* elemenata. Sličan postupak se nastavlja sve dok se ne upiše element n. Zbir operacija je: $k + k + k + 2k + k + 3k + ...$ Da bi se smestilo n elemenata, realokaciju je potrebno vršiti oko n/k puta, pa će ukupan broj operacija biti otprilike jednak:

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

Ukupan broj upisa asimptotski je jednak $\frac{n^2}{2k}$ tj. $O(n^2)$ i stoga je amortizovana cena jedne operacije asimptotski jednaka $\frac{n}{2k}$, što je $O(n)$.
Druga strategija može biti geometrijska i podrazumeva da se svaki put veličina niza poveća q puta za neki faktor $q > 1$. Pretpostavimo da je početna veličina niza m_0. Nakon početne alokacije možemo da upišemo m_0 elemenata. Nakon toga se vrši prva realokacija u kojoj se veličina niza povećava na qm_0 elemenata i pri čemu se prepisuje m_0 elemenata. Nakon toga se vrši upis narednih qm_0-m_0 elemenata. U narednoj realokaciji veličina niza se povećava na q^2m_0 i se prepisuje qm_0 elemenata. Upisuje se preostalih q^2m_0-qm_0 elemanata. Ukupan broj upisa u niz je: $m_0+m_0+(qm_0-m_0)+qm_0+(q^2m_0-qm_0)+...= m_0+qm_0+q^2m_0+...$ Posle r realokacija broj upisa je: $m_0(1+...+q^r) = m_0(q^{r+1}-1)/(q-1)$. Ako pretpostavimo da je ceo niz popunjen posle r realokacija, tj. da je $n = m_0q^r$, onda je ukupan broj operacija potrebnih za popunjavanje niza jednak: $m_0(q\frac{n}{m} - 1)/(q-1) = (qn - m_0)/(q-1)$. $O(n)$, amortizovano $O(1)$.

2.9 Zamena iteracije eksplicitnom formulom - primeri

Zbir prvih n prirodnih brojeva
Problem: Definisati efikasan algoritam koji izračunava zbir 1 + 2 +...+ n.
Jedno rešenje je da se zadatak reši u složenosti *O(n)*, tako što se primeni petlja i algoritam sabiranja serije elemenata. Ovo rešenje je neefikasno, jer se do rešenja može doći u složenosti *O(1)* korišćenje eksplicitne, Gausove formule.

```
int zbir = n*(n+1)/2;
```

Prilikom implementacije treba voditi računa i o prekoračenju i malo bolja implementacija prvo deli, pa onda množi.

```
int zbir = n % 2 == 0 ? (n/2)*(n+1) : ((n+1)/2)*n;
```

Nedostajući broj
Problem: Dat je niz od n elemenata koji sadrži različite brojeve iz skupa 0, 1, 2,..., n (tačno jedan broj nedostaje). Odredi koji broj nedostaje.
Jedno rešenje može biti zasnovano na linearnoj pretrazi svih kandidata. Za sve brojeve od *0* do n proveravamo da li su sadržani u nizu. Linearna pretraga niza od n elemenata u najgorem slučaju zahteva *O(n)* koraka, pa pošto se traži *n+1* element, složenost je $O(n^2)$.
Bolje rešenje može biti zasnovano na sortiranju elemenata (*O(nlogn)*) i zatim linearnoj proveri svake pozicije da li je $a_i = i$. Prva koja nije ukazuje na nedostajući broj. Tu poziciju možemo identifikovati i binarnom pretragom, ali algoritmom i dalje dominira složenost sortiranja. Zbir svih elemenata iz skupa 0, 1, 2,..., n je *n(n+1)/2*. To je zbir elemenata koji se nalaze u nizu i nedostajućeg elementa. Nedostajući element je, dakle, jednak, razlici između *n(n+1)/2* i zbira svih elemenata niza koji lako možemo izračunati u vremenu *O(n)*.

Broj deljivih u intervalu
Problem: Definisati efikasan algoritam koji određuje koliko je brojeva u intervalu [a, b], $0 \leq a \leq b$ deljivo datim brojem k.
Naivan način da se ovaj zadatak reši je da se primeni linearni prolaz kroz interval, da se proveri deljivost svakog elementa i brojanje onih koji zadovoljavaju traženi uslov. Složenost ovog algoritma, jasno je *O(b - a)*.
Da bi broj x bio deljiv brojem k potrebno je da postoji neko n tako da je $x = n \cdot k$. Pošto x mora biti u intervalu [a, b], mora da važi da je $a \leq n \cdot k$ i $n \cdot k \leq b$. Najmanje n koje zadovoljava prvu nejednačinu jednako je $n_l = [a/k]$. Najveće n koje zadovoljava drugu nejednačinu jednako je $n_d = [b/k]$. Bilo koji broj iz intervala $[n_l, n_d]$ zadovoljava obe nejednakosti i predstavlja količnik nekog broja iz intervala [a, b] brojem k. Slično, bilo koji broj iz intervala [a, b] deljiv brojem k daje neki količnik iz intervala $[n_l, n_d]$. Traženi broj brojeva iz intervala [a, b] koji su deljivi brojem k je broj brojeva u intervalu $[n_l, n_d]$ a to je $n_d - n_l + 1$ ako je $n_d \geq n_l$, tj. 0 ako je taj interval prazan. Složenost ovakvog algoritma je *O(1)*.

```
int nl = a % k == 0 ? a/k : a/k + 1; // ceil(a/k)
int nd = b/k; // floor(b/k)
int broj = nd >= nl ? nd-nl+1 : 0;
```

2.10 Eliminisanje identičnih izračunavanja - primeri

Fibonačijevi brojevi

Problem: Fibonačijev niz 0, 1, 1, 2, 3, 5, 8, 13,... je takav da je zbir njegova dva susedna elementa uvek daje element koji sledi iza njih. Napisati program koji izračunava njegov član na poziciji *n*.

Rekurzivno rešenje je moguće implementirati veoma jednostavno. Jednačina koja određuje složenost ove funkcije je T(n) = T(n-1)+T(n-2)+1, T(1) = T(0) = 0. Njeno rešenje je eksponecijalno i funkcija je izrazito neefikasna. Problem je u tome što se iste vrednosti računaju više puta.

U slučajevima preklapajućih rekurzivnih poziva, kao što je ovaj, može se primeniti tehnika *dinamičkog programiranja*. Osnovna ideja je da se rezultati rekurzivnih poziva pamte, da se svaki put pre ulaska u rekurziju proveri da li je taj poziv već izvršavan i ako jeste, da se vrati njegova već izračunata vrednost. Ova tehnika se naziva *memoizacija*. Već ovom transformacijom složenost je svedena na *O(n)*.

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

U svakom trenutku se koriste samo dva prethodna elementa niza, pa je ceo niz moguće eliminisati i tako dobiti program čija je vremska složenost *O(n)*, a memorijska *O(1)*.

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

Najmanja tačna perioda niza

Problem: Dat je niz a dužine 2^k. Broj p je tačna perioda niza a ako se niz a može dobiti ponavljanjem podniza prvih p elemenata niza, bez dodatnih elemenata.

Jedan pristup rešavanju zadatka je da se isprobaju redom sve moguće dužine perioda od 1 do dužine niza, koji su stepeni dvojke i da se prijavi prvi broj za koji se ustanovi da predstavlja dužinu periode. Ako je niz periodičan, tada je $a_0 = a_p = a_{2p} = ...$, $a_1 = a_{p+1} = a_{2p+1} = ...$ itd. Za svaku poziciju i mora da važi da je $a_i = a_i mod p$. Na osnovu toga možemo jednostavno napraviti funkciju koja proverava da li je p perioda niza (za p koje deli n. Složenost najgoreg slučaja ovog rešenja je *O(nlogn)*.

```
int jePerioda(int a[], int n, int p) {
    for (int i = p; i < n; i++)
        if (a[i] != a[i % p])
            return false;
    return true;
}

int minPerioda(int a[], int n) {
    for (int p = 1; p <= n; p *= 2)
        if (jePerioda(a, n, p))
            return p;
    return n;
}
```

Efikasniji način: Ključni uvid je to da ako je p perioda niza, tada je i $2p$ takođe perioda niza. Dužina niza n je sigurno perioda niza. Ako prva polovina niza nije jednaka drugoj, tada je najmanja perioda dužina niza. Ako se ustanovi da je prva polovina niza jednaka drugoj, tada je perioda sigurno i $n/2$, što je dužina prve polovine niza. Određivanje najkraće periode celog niza se tada svodi na određivanje najkraće periode prve polovine niza. Ovim smo dobili induktivno rekurzivnu konstrukciju koja se završava kada je tekuća dužina niza 1 ili kada se naiđe na niz čija prva polovina nije jednaka drugoj. Rekurzivna implementacija zadovoljava rekurentnu jednačinu $T(n) = T(n/2) + O(n)$, čije je rešenje *O(n)*.

```
int minPeriod(int a[], int n) {
    if (n == 1)
        return 1;
    for (int i = 0; i < n / 2; i++)
        if (a[i] != a[n/2 + i])
            return n;
    return minPeriod(a, n/2);
}
```

2.11 Eliminisanje nepotrebnih izračunavanja - primeri

Drugi po veličini element u nizu
Problem: Data je lista poena studenata nakon prijemog. Koliko poena je imao drugi student na rang listi?
Naivno rešenje podrazumeva da se niz sortira opadajuće, pa da se pročita element na drugoj poziciji. Složenost ovog rešenja dolazi od složenosti sortiranja i iznosi *O(nlogn)*.
Jedan od načina da se složenost smanji na *O(n)* je da se primene samo prve dve runde algoritma sortiranja selekcijom.

```
int drugiNaListi(int a[], int n) {
    for (int i = 0; i < 2; i++) {
        int max_p = i
        for (int j = i + 1; j < n; j++)
            if (a[j] > a[max_p])
                max_p = j;
        swap(a[i], a[max_p]);
    }
    return a[1];
}
```

Najbolje rešenje je koristi specijalizovani algoritam:

```
int drugiNaListi(int a[], int n) {
    int prviMax, drugiMax;
    prviMax = drugiMax = -1;
    for(int i = 0; i < n; i++) {
        if (a[i] > prviMax) {
            drugiMax = prviMax;
            prviMax = a[i];
        }
        else if (a[i] > drugiMax)
            drugiMax = a[i];
    }
    return drugiMax;
}
```

Morzeov niz
Problem: Niz koji se sastoji od nula i jedinica, gradi se na sledeći način: prvi element je 1; drugi se dobija logičkom negacijom prvog, treći i četvri logičkom negacijom prethodna dva, peti, šesti, sedmi i osmi logičkom negacijom prva četiri itd. Dakle, krenuvši od jednočlanog segmenta 1, svakom početnom segmentu koji je dužine 2^k dopisuje se segment iste dužine dobijen logičkom negacijom svih elemenata početnog segmenta. Definisati funkciju koja za dato n određuje n-ti član niza (brojanje kreće od 1).

Direktan način da se zadatak reši je da se efektivno popuni niz sve do pozicije n i da se pročita element na mestu n. Jedan način da se to uradi je da se upotrebi ugnežđena petlja gde se u svakom unutrašnjem koraku duplira dužina niza. Umesto dvostruke, možemo upotrebiti i jednostruku petlju. Složenost ovog pristupa je *O(n)*. Prethodno rešenje podrazumeva formiranje svih elemenata niza koji prethode n-tom članu.
Problem možemo rešiti mnogo efikasnije induktivno-rekurzivnom konstrukcijom. Bazu inducije jasno predstavlja slučaj $m_1 = 1$. Prilikom negiranja početnog segmenta dobijamo da je $m_2 = !m_1$. Dakle, u ovom slučaju važi da je $m_n = !m_{n-1}$. Negiranjem narednog segmenta dobijamo da je $m_3 = !m_1$ i $m_4 = !m_2$. U ovom slučaju važi da je $m_n = !m_{n-2}$. Za $n > 1$ važi rekurentna formula $m_n = !m_{n-k}$, gde je k maksimalni stepen broja 2 koji je strogo manji od n. Ova rekurentna formula omogućava veoma efikasno izračunavanje traženog člana niza. Na primer, m_{15}=!m_{15-8}=!m_7=!(!m_{7-4})=m_{7-4}=m_3=!m_{3-2}=!m_1=!1=0.

```
int maxStepen2(int n) {
    int max = 1;
    while ((max << 1) < n)
        max <<= 1;
    return max;
}
int morzeov(int n) {
    if (n == 1)
        return 1;
    return !morzeov(n - maxStepen2(n));
}
```

Složenost ovako implementirane funkcije za određivanje najvećeg stepena dvojke koji je strogo manji od n je *O(logn)*. Složenost je u najgorem slučaju $O(log^2(n))$.

2.12 Inkrementalnost - primeri

Svi faktorijeli
Problem: Napisati program koji ispisuje vrednosti svih faktorijela od 1 do n. Naivan način je da se implementira funkcija koja izračunava vrednost faktorijela i da se ona u petlji poziva, $O(n^2)$.
Mnogo bolje rešenje je da se primeti da je *k! = k(k - 1)!*. Zato faktorijele ne treba izračunavati jedan po jedan već svaki sledeći treba izračunati na osnovu prethodnog. Faktorijeli zapravo čine rekurentno definisanu seriju 0! = 1 i *k! = k(k - 1)!*, za $k > 0$. Broj množenja je *n-1* pa je složenost algoritma *O(n)*.

```
int n, p = 1;
cin >> n;
for(int k = 1; k <= n; k++) {
    p *= k;
    cout << p << endl;
}
```

Maksimalni zbir segmenta
Problem: Definisati efikasnu funkciju koja pronalazi najveći mogući zbir segmenta (podniza uzastopnih elemenata) datog niza brojeva.
Najdirektniji mogući način da se zadatak reši je da se izračuna zbir svih segmenata.

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

Unutrašnja petlja se izvršava *j-i+1* puta, što odgovara dužini segmenta. Spoljne petlje koje nabrajaju sve segmente nabrajaju jedan segment dužine n, dva segmenta dužine *n-1*, tri segmenta dužine *n-2*, ... i n segmenata dužine 1. Znači da je broj koraka jednak 1·n + 2·(n - 1) + 3·(n - 2) + ... + (n - 1)·2 + n·1. Dakle, segmenata dužine k ima *n-k+1*, pa važi da je prethodni zbir jednak:

$$\sum_{k=1}^{n} k(n-k+1) = (n+1)\sum_{k=0}^{n} k - \sum_{k=0}^{n} k^2 = (n+1)\frac{n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6}$$

Ovaj algoritam je veoma naivan i složenosti je $O(n^3)$.
Složenost se može smanjiti ako se primeti da se u većini slučajeva naredni segment dobija od prethodnog tako što se prethodni segment proširi za jedan element zdesna. Umesto da zbir proširenog segmenta svaki put računamo iznova, možemo iskoristiti to što već znamo zbir prethodnog segmenta i njega možemo samo uvećati za novododati element.

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

Za svako i unutrašnja petlja se izvršava *n-i* puta i u svakom koraku se vrši jedno sabiranje. Složenost je $O(n^2)$.

Određivanje broja rastućih segmenata datog niza brojeva
Problem: Dat je niz a celih brojeva. Definisati efikasan algoritam kojim se određuje koliko u tom nizu postoji rastućih segmenata.
Najdirektnije rešenje je rešenje grubom silom i zasniva se na tome da se za svaki segment u nizu eksplicitno proveri da li je rastući. Provera da li je segment rastući može se zasnovati na linearnoj pretrazi za dva susedna elementa u kojima je $a_i \geq a_{i-1}$; ako takva dva elementa ne postoje, niz je rastući. Ovo rešenje je veoma neefikasno. Postoji $O(n^2)$ segmenata, a monotonost svakog se proverava algoritmom linearne složenosti, pa je ukupna složenost najgoreg slučaja $O(n^3)$. Do rešenja možemo efikasnije doći ako primenimo svojstvo inkrementalnosti tj. činjenicu da ako znamo da je segment a_i,..., a_j rastući, tada je segment a_i,..., a_j, a_{j+1} rastući ako i samo ako je $a_j < a_{j+1}$. Dodatno, ako taj segment nije rastući, znamo da rastući ne može biti ni jedan dalji segment koji počinje na poziciji i, tako da unutrašnju petlju možemo prekinuti čim se naiđe na dva uzastopna elementa koji nisu u rastućem redosledu. Složenost ovog algoritma je $O(n^2)$.

```
int brojRastucih = 0;
for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n && a[j] > a[j-1]; j++)
        brojRastucih++;
```

Najveći težinski zbir rotacija niza
Problem: Dat je niz a celih brojeva dužine n. Dozvoljena je operacija cikličnog pomeranja tj. rotacije niza ulevo za proizvoljan broj mesta. Napisati program kojim se određuje najmanji broj pomeranja ulevo tako da težinski zbir u transformisanom nizu ima najveću vrednost.

$$\sum_{i=0}^{n-1} i a_i = 0a_0 + 1a_1 + ... + (n-1)a_{n-1}$$

Zadatak možemo rešiti tako što *n-1* put niz efektivno rotiramo za po jedno mesto ulevo izračunavajući svaki put traženi težinski zbir iznova, tražeći maksimum i poziciju maksimuma tako dobijenih težinskih zbirova. Rotaciju možemo realizovali bibliotečkom funkcijom za rotiranje (složenost joj je $O(n)$) ili sami isprogramirati. Pošto broj operacija potreban za nezavisno izračunavanje svakog težinskog zbira linearno zavisi od dužine niza n, složenosti je $O(n^2)$.
Umesto efektivne rotacije svih elemenata niza, efekat obilaska niza koji je rotiran za k mesta ulevo možemo postići tako što obilazak krećemo od pozicije k, a zatim u petlji koja ima n iteracija uvećavamo brojač za 1, ali po modulu n. Dobitak na brzini može biti značajan, ali složenost i dalje ostaje $O(n^2)$.
Još jedna tehnika kojom možemo izbeći rotaciju je da alociramo dvostruko više memorije, da elemente originalnog niza smestimo dva puta, jedan iza drugog i da zatim efekat rotacije za k mesta postižemo tako što obilazimo n elemenata tako proširenog niza počevši od pozicije k. Ovim dobijamo na brzini, ali gubimo na zauzeću memorije. I ovaj put dobitak na brzini može biti značajan, ali složenost i dalje ostaje $O(n^2)$.

Možemo uočiti da je u traženom zbiru posle pomeranja ulevo svaki element niza, izuzev prvog elementa a_0, jedan put manje uključen nego pre pomeranja, a prvi element je uključen *n-1* put. Obeležimo sa z_i traženi zbir dobijen prilikom pomeranja polaznog niza ulevo i puta, a sa z zbir svih elemenata niza. Prema tome važe sledeće jednakosti:

$$z_0 = 0a_0 + 1a_1 + 2a_2 + ... + (n-2)a_{n-2} + (n-1)a_{n-1}$$

$$z_1 = 0a_1 + 1a_2 + 2a_3 + ... + (n-1)a_{n-2} + (n-1)a_0$$

$$...$$

$$z_{n-1} = 0a_{n-1} + 1a_0 + 2a_1 + ... + (n-2)a_{n-3} + (n-1)a_{n-2}$$

$$z = a_0 + a_1 + ... + a_{n2} + a_{n1}$$

Vazi: z_{n-2} - z_{n-1} = a_{n-1} + a_0 + ... + a_{n-3} - *(n - 1)*a_{n-2} = z - na_{n-2}. Prema tome vazi: z_i = z_{i-1} - z + na_{i-1}.
Vreme potrebno za izračunavanje početnih zbirova z_0 i z linearno zavisi od dužine niza n, dok je za izračunavanje svakog od narednih *n-1* zbirova dovoljan konstantan broj operacija, tako da je ukupna vremenska složenost algoritma linearna tj. *O(n)*.

```
int brojRotacijaZaMaksimalniZbir(const vector<int>& a) {
    int tezinskiZbir = 0;
    int zbir = 0;
    for (int i = 0; i < n; i++) {
        tezinskiZbir += i*a[i];
        zbir += a[i];
    }
    int maksTezinskiZbir = tezinskiZbir;
    int maksBrojRotacija = 0;
    for (int i = 1; i < n; i++) {
        tezinskiZbir = tezinskiZbir - zbir + n * a[i-1];
        if (tezinskiZbir > maksTezinskiZbir) {
            maksTezinskiZbir = tezinskiZbir;
            maksBrojRotacija = i;
        }
    }
    return maksBrojRotacija;
}
```

Postavljanje rutera
Problem: Duž jedne ulice su ravmnomerno raspoređene zgrade (rastojanje između svake dve susedne je jednako). Za svaku zgradu je poznat broj korisnika koje novi dobavljač interneta treba da poveže. Odrediti u koju od zgrada treba postaviti ruter tako da bi ukupna dužina optičkih kablova kojim se svaki od korisnika povezuje sa ruterom bila minimalna.
Naivno rešenje bi podrazumevalo da se izračuna dužina kablova za svaku moguću poziciju rutera i da se odabere najmanji. Da bismo izračunali dužinu kablova,

ako je ruter u zgradi na poziciji k, računamo zapravo zbir $\sum_{i=0}^{n-1} |k-i|a_i$, gde je a_i broj korisnika u zgradi i. Tu težinsku sumu možemo izračunati u vremenu $O(n)$, pa pošto se ispituje n pozicija, algoritam bi bio složenosti $O(n^2)$.
Mnogo bolje rešenje i linearni algoritam možemo dobiti ako primenimo princip inkrementalnosti. Razmotrimo kako se dužina kablova menja kada se ruter pomera sa zgrade k na zgradu $k+1$. Ako je ruter na zgradi k tada je dužina kablova jednaka: $\sum_{i=0}^{k-1}(k-i)a_i + \sum_{i=k+1}^{n-1}(i-k)a_i$. Ako je ruter na zgradi $k+1$, tada je dužina kablova jednaka: $\sum_{i=0}^{k}(k+1-i)a_i + \sum_{i=k+2}^{n-1}(i-k-1)a_i$. Razlika između te dve sume jednaka je: $\sum_{i=0}^{k} a_i - \sum_{i=k+1}^{n-1} a_i$.
Dužinu kablova za ruter u zgradi $k+1$ dobijamo od dužine kablova za ruter u zgradi k tako što tu dužinu uvećamo za ukupan broj stanara zaključno sa zgradom k i umanjimo je za ukupan broj stanara počevši od zgrade $k+1$. Pomeranjem rutera za dužinu jedne zgrade nadesno, svakom stanaru koji živi zaključno do zgrade k dužina kabla se povećala za jedno rastojanje između zgrada, a svim stanarima od zgrade $k+1$ nadesno se ta dužina smanjuje za jedno rastojanje između zgrada. Ukupne brojeve stanara pre i posle date zgrade možemo takođe računati inkrementalno. Složenost ovog algoritma je $O(n)$.

```
long long duzina_kablova = 0;
for (int i = 0; i < n; i++)
    duzina_kablova += stanara[i] * i;
long long stanara_pre = 0;
long long stanara_posle = 0;
for (int i = 0; i < n; i++)
    stanar_posle += stanara[i];
long long min_duzina_kablova = duzina_kablova;
for (int k = 1; k < n; k++) {
    stanara_pre += stanara[k-1];
    stanara_posle -= stanara[k-1];
    duzina_kablova += stanara_pre - stanara_posle;
    if (duzina_kablova < min_duzina_kablova)
        min_duzina_kablova = duzina_kablova;
}
cout << min_duzina_kablova << endl;
```

2.13 Odsecanje u pretrazi - primeri

Ispitivanje da li je broj prost
Problem: Definisati efikasnu funkciju koja proverava da li je broj prost.
Naivno rešenje je zasnovano na linearnoj proveri svih delilaca. Složenost najgoreg slučaja ovog algoritma je očigledno O(n) (i ona se dobija kada je broj prost).
Na osnovu teoreme koja kaže da ako broj n ima delioca d koji je veći ili jednak $\sqrt{n}$ onda sigurno ima delioca koji je manji ili jednak $\sqrt{n}$ (to je broj n/d), broj kandidata za delioce možemo značajno smanjiti i dobiti efikasniji algoritam. Složenost najgoreg slučaja ovog algoritma je $O(\sqrt{n})$.

```
bool jeProst(unsigned n) {
    if (n == 1)
        return false;
    for (unsigned d = 2; d*d <= n; d++)
        if (n % d == 0)
            return false;
    return true;
}
```

Prethodna implementacija se može malo ubrzati, na osnovu činjenice da su svi prosti brojevi oblika *6k+1* ili *6k-1*, pa se petlja može organizovati tako što se brojač uvećava za 6, a u telu se ispituju dva kandidata. Nema potrebe posebno proveravati da li je broj n oblika *6k+1* ili *6k-1*, jer početna provera deljivosti sa 2 i sa 3 eliminiše sve brojeve koji nisu tog oblika. Složenost je i dalje $O(\sqrt{n})$.

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

Binarna pretraga

Problem: Proveriti da li u strogo rastuće sortiranom nizu brojeva postoji neka pozicija i takva da je $a_i = i$.

Direktno rešenje je zasnovano na linearnoj pretrazi i ne koristi činjenicu da je niz sortiran.

Mnogo efikasnije rešenje zasnovano je na binarnoj pretrazi. Ako je $a_i = i$, tada je a_i - $i = 0$. Pokažimo da je niz a_i - i neopadajući. Posmtarajmo dva elementa a_i i a_j na pozicijama na kojima je $0 \leq i < j < n$. Pošto je niz a strogo rastući, važi da je $a_{i+1} > a_i$, pa je $a_{i+1} \geq a_i + 1$. Nastavljanjem ovog rezona važi da je $a_j \geq a_i + j$. Zato je a_j - $j \geq a_i \geq a_i$ - i. Rešenje, dakle, možemo odrediti tako što binarnom pretragom proverimo da li niz a_i - i sadrži nulu i ako sadrži, tada je rešenje prva pozicija na kojoj se ta nula nalazi.

```
int fiksnaTacka(int[] a, int n) {
    int l = 0, d = n-1;
    while (l <= d) {
        int s = l + (d - l) / 2;
        if (a[s] < s) l = s + 1;
        else  d = s - 1;
    }
    if (l < n && a[l] == l) return l;
    else  return -1;
}
```

2.14 Maksimalni zbir segmenta - odsecanje

Veliki broj segmenata uopšte ne moramo da obrađujemo, jer iz nekih drugih razloga znamo da njihov zbir ne može biti maksimalan.
Razmotrimo bilo koji niz koji počinje negativnim brojem. Nijedan segment koji počinje od tog broja, ne može biti segment maksimalnog zbira, pošto se izostavljanjam prvog broja dobija veći zbir. Ovo svojstvo je i opštije. Ukoliko niz počinje prefiksom negativnog zbira, iz istog razloga, nijedan segment čiji je on prefiks ne može biti segment maksimalnog zbira. Otud, pri inkrementalnom proširivanju intervala udesno, čim se ustanovi da je tekući zbir negativan, moguće je prekinuti dalje proširivanje. Iako se na ovaj način može preskočiti razmatranje nekih segmenata, u najgorem slučaju složenost nije smanjena. Na primer, u slučaju da su elementi niza strogo pozitivni, zbir nikad ne postaje negativan i izvršavanje je i dalje kvadratne složenosti. Zahvaljujući ovom zapažanju, nije neophodno uvećavati promenljivu i za jedan, već je moguće nastaviti iza elementa čijim je uključivanjem suma postala negativna. Složenost ovog algoritma je $O(n)$.

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

2.15 Pretprocesiranje radi efikasnije pretrage - primeri

Provera duplikata u nizu
Problem: Definisati funkciju koja efikasno proverava da li u datom nizu celih brojeva ima duplikata.
Naivni način je da se ispita svaki par elemenata. Složenost najgoreg slučaja ovog pristupa je prilično očigledno kvadratna.
Efikasniji algoritam dobijamo ako prvo niz sortiramo. Ako u nizu ima duplikata, nakon sortiranja oni će se naći na susednim pozicijama, pa da bismo proverili duplikate možemo samo proveriti susedne elemente niza. Sortiranje smo sproveli korišćenjem bibliotečke funkcije i njena složenost je $O(nlogn)$. Nakon toga, pretraga se vrši u vremenu $O(n)$. Dakle, ukupna složenost je veća od ove dve, a to je $O(nlogn)$.

```
bool imaDuplikata(int a[], int n) {
    sort(a, a+n);
    for (int i = 0; i < n-1; i++)
        if (a[i] == a[i+1])
            return true;
    return false
}
```

Zbirovi segmenata

Problem: Dat je niz celih brojeva dužine n. Napisati program koji izračunava zbirove m njegovih segmenata određenih intervalima pozicija $[l_i, d_i]$.

Direktan način je da se nakon učitavanja niza za svaki segment zbir iznova računa u petlji. Ako niz ima n elemenata i želimo da izračunamo zbirove njegovih m segmenata, ukupna složenost bila bi $O(n \cdot m)$.

Jedan od načina da se zadatak efikasno reši je to da se primeti da se svaki zbir segmenta može predstaviti kao razlika dva zbira prefiksa niza. Dakle, ako znamo zbirove svih prefiksa, zbir svakog segmenta možemo izračunati u vremenu $O(1)$. Zbirove prefiksa, naravno, možemo računati inkrementalno, tako da je vreme potrebno za njihovo izračunavanje $O(n)$. Ukupna složenost je onda $O(n + m)$.

```
int n;
cin >> n;
vector<int> zbirPrefiksa(n+1);
zbirPrefiksa[0] = 0;
for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    zbirPrefiksa[i+1] = zbirPrefiksa[i] + x;
}
int m;
cin >> m;
for (int i = 0; i < m; i++) {
    int l, d;
    cin >> l >> d;
    cout << zbirPrefiksa[d+1] - zbirPrefiksa[l] << endl;
}
```

3 Induktivno-rekurzivna konstrukcija

3.1 Induktivno-rekurzivna konstrukcija - Grejovi kodovi

Problem: Grejov kôd dužine 2^n definiše se kao niz koji sadrži sve zapise n-tocifrenih binarnih brojeva takve da se svaka dva susedna zapisa (kao i prvi i poslednji zapis) razlikuju tačno u jednom bitu. Definisati funkciju koja konstruiše Grejov kôd dužine 2^n za proizvoljno n.
Opšta shema koja se može upotrebiti za dobijanje Grejovog koda:

- Ako je dužina Grejovog koda 0, taj kôd sadrži samo praznu nisku.
- Ako je dužina Grejovog koda 2^n tada se Grejov kôd dobija tako što se poređaju redom elementi Grejovog koda dužine 2^{n-1} prošireni početnom nulom, a zatim se u obratnom redosledu poređaju redom elementi Grejovog koda dužine 2^{n-1} prošireni početnom jedinicom.

Dokaz da prethodna procedura daje Grejov može jednostavno izvesti matematičkom inducijom.
Definišimo funkciju koja određuje k-ti po redu zapis Grejovog koda dužine 2^n ($0 \leq k < 2^n$). Na osnovu prethodnog razmatranja nju je veoma jednostavno definisati rekurzivno. Ako je n nula, u pitanju je prazna niska, a u suprotnom razlikujemo slučaj kada je $0 \leq k < 2^{n-1}$ u kom se vraća k-ti element Grejovog koda dužine 2^{n-1} dopunjen početnom nulom i slučaj $2^{n-1} \leq k < 2^n$ u kom se vraća $2^n - 1 - k$-ti element Grejovog koda dužine 2^{n-1} dopunjen početnom jedinicom. Izračunavanje stepena dvojke trivijalno se vrši bitovskim operacijama.

```
string grej(int n, int k) {  // O(n)
    if (n == 0)
        return "";
    if (k < (1 << (n - 1)))
        return "0" + grej(n - 1, k);
    else
        return "1" + grej(n - 1, (1 << n) - 1 - k);
}
```

Invarijanta petlje kod iterativne implementacije je da je $n_0 \geq n \geq 0$ i da se u promenljivoj *rez* nalazi prefiks dužine $n_0 - n$ traženog Grejovog koda (u svakom koraku petlje taj prefiks produžavamo za jedan karakter).
Grejov kôd se može izračunati i direktno, ako se predstavi u obliku neoznačenog broja (zbog vodećih nula dužina tada nije bitna).

```
unsigned grej(unsigned k) {
    return k ^ (k >> 1);
}
```

3.2 Induktivno-rekurzivna konstrukcija - prefiksni na osnovu infiksnog i postfiksnog obilaska stabla

Problem: Poznati su infiksni i postfiksni obilazak binarnog drveta koje nije nužno uređeno (čvorovi drveta sadrže karaktere i obilasci su zadati pomoću dve niske iste dužine). Napisati program koji na osnovu njih određuje prefiksni obilazak. Pretpostaviti da svi čvorovi u drvetu sadrže različite vrednosti.
Na primer, postfiksni obilazak je 34268751 a infiksni 32416578. Potrebno je da rekonstruišemo prefiksni obilazak koji je u ovom slučaju 12345678.
Ključni uvid za rešenje zadatka je taj da koren drveta možemo jednostavno odrediti kao poslednji element u postfiksnom obilasku. Tada taj element možemo pronaći i u infiksnom obilasku i na osnovu toga odrediti koji elementi pripadaju levom, a koji desnom poddrvetu. Znajući broj elemenata u svakom poddrvetu, možemo pročitati i njihove prefiksne obilaske. Dakle, veoma jednostavno možemo odrediti koren drveta, infiksni i postfiksni obilazak levog poddrveta i infiksni i postfiksni obilazak desnog poddrveta. Ovim smo problem sveli na dva manja potproblema koja se mogu rešavati rekurzivno. Bazu predstavlja slučaj praznog stabla (kada su sva tri obilaska prazna).

```
string nadjiPre(const string& post, const string& in) {
    if (in == "" && post == "")
        return "";
    char koren = post[post.size() - 1];
    size_t levo = in.find(koren);
    size_t desno = in.size() - levo - 1;
    string in_l = in.substr(0, levo);
    string in_d = in.substr(levo + 1, desno);
    string post_l = post.substr(0, levo);
    string post_d = post.substr(levo, desno);
    return koren + nadjiPre(post_l, in_l) + nadjiPre(post_d, in_d);
}
```

3.3 Induktivno-rekurzivna konstrukcija - rotiranje niza za k mesta

Problem: Neka je dat niz od n elemenata. Definisati funkciju složenosti $O(n)$ koja njegov sadržaj rotira za k mesta ulevo, bez korišćenja pomoćnog niza.
Jedno, naivno rešenje bilo bi da se niz k puta rotira za po jedno mesto u levo, no to bi bilo prilično neefikasno ($O(kn)$).
Primetimo da uvek možemo da obezbedimo da je $k < n$, jer rotacija za n mesta vraća niz u početnu poziciju, tako da umesto rotacije za k mesta možemo da uradimo rotaciju za broj koji se dobije kao ostatak pri deljenju k sa n.
Neka niz ima n elemenata i neka ga rotiramo za k mesta ulevo. Neka prvih k elemenata čine blok koji ćemo označiti sa L, a preostalih n - k elemenata čine blok koji ćemo označiti sa D. Razmotrimo sledeće slučajeve, u zavisnosti od odnosa dužina ta dva bloka.

• Ako je blokovi L i D jednake dužine, njihovom razmenom se dobija traženo rešenje.
• Ako je blok L kraći, označimo sa D_1 početni deo bloka D dužine k, a sa D_2, preostali deo bloka D. Elementi bloka D_1 treba da budu početni elementi u traženom rešenju i da bismo to postigli, možemo ih razmeniti sa elementima bloka L. Time iz situacije LD_1D_2 dolazimo u situaciju D_1LD_2, dok je traženo rešenje oblika DL, tj. D_1D_2L. Da bismo to postigli, potrebno je da niz LD_2 zarotiramo za dužinu bloka L ulevo (a to je opet k), što je problem istog oblika, ali manje dimenzije od polaznog.
• Ako je blok L duži, označimo sa L_1 početni deo bloka L dužine n - k, a sa L_2, preostali deo bloka L. Elementi bloka D treba da budu početni elementi u traženom rešenju i da bismo to postigli, možemo ih razmeniti sa elementima bloka L_1. Time iz situacije L_1L_2D dolazimo u situaciju DL_2L_1, dok je traženo rešenje oblika DL, tj. DL_1L_2. Da bismo to postigli, potrebno je da niz L_2L_1 zarotiramo za dužinu bloka L_2 ulevo (a to je 2k - n), što je problem istog oblika, ali manje dimenzije od polaznog.

```
void razmeni(int a[], int p1, int p2, int m) { // O(m)
    for (int i = 0; i < m; i++)
        swap(a[p1+i], a[p2+i]);
}

void razmeniBlokove(int a[], int p1, int d1, int p2, int d2) {
    if (d1 == 0 || d2 == 0)
        return;
    if (d1 <= d2) {
        razmeni(a, p1, p2, d1);
        razmeniBlokove(a, p1 + d1, d1, p2 + d1, d2 - d1);
    }
    else {
        razmeni(a, p1, p2, d2);
        razmeniBlokove(a, p1 + d2, d1 - d2, p2, d2);
    }
}
```

Centralna mera progresa u algoritmu je zbir dužina blokova koji se razmenjuju. Ona kreće od n i smanjuje se sve dok ne dođe do nule. U prvom slučaju se vrši razmena blokova dužine d_1 za šta je potrebno $O(d_1)$ koraka i nakon toga se vrši rekurzivni poziv takav da je zbir dužina blokova upravo za d_1 manji od polaznog zbira dužina. Slično, u drugom slučaju se vrši zamena blokova dužine d_2 za šta je potrebno $O(d_2)$ koraka i nakon toga se vrši rekurzivni poziv takav da je zbir dužina blokova upravo za d_1 manji od polaznog zbira dužina. Dakle, rekurentna jednačina je u oba slučaja jednaka $T(n) = T(n - d) + O(d)$ i njeno rešenje je $T(n) = O(n)$.
Ovaj algoritam možemo izraziti veoma jednostavno i iterativno (rekurzija je repna, pa se može lako eliminisati). Mozemo ukloniti i pomoćnu funkciju *razmeni* i dobiti veoma elegantan i efikasan algoritam.

```
k %= n;
int l = 0;
int d = k;
while (l != k && d != n) { // invarijanta: 0 <= l <= k <= d <= n
    swap(a[l++], a[d++]);
    if (l == k)
        k = d;
    if (d == n)
        d = k;
}
```

3.4 Induktivno-rekurzivna konstrukcija - određivanje zvezde

Problem: U nekoj grupi ljudi osoba se naziva zvezda (engl. superstar) ako je svi prisutni znaju, a ona ne poznaje nikoga od prisutnih. Definisati funkciju za ispitivanje da li u datom skupu ljudi postoji zvezda. Data je matrica logičkih vrednosti kojom se određuje ko koga poznaje (na poziciji (i, j) se nalazi vrednost tačno akko osoba i poznaje osobu j).
Direktan način je da za svaku osobu proverimo da li zadovoljava uslov zvezde. Složenost najgoreg slučaja ovog algoritma je $O(n^2)$, mada se ta složenost veoma teško dostiže.
Poboljšanje možemo pokušati induktivno-rekurzivnim pristupom. Prva ideja je da problem pronalaženja zvezde u skupu od n osoba svodimo na problem pronalaženja zvezde u skupu od *n-1* osoba. Složenost najgoreg slučaja takvog algoritma je $T(n) = T(n-1) + O(n)$, što je $O(n^2)$.
Ideja rešenja je da se problem posmatra "unazad". Broj osoba koje nisu zvezde je sigurno mnogo veći od broja osoba koje jesu zvezde, pa je identifikovanje ne-zvezde mnogo jednostavnije od identifikovanja zvezde. Ključna ideja ove induktivne konstrukcije je da veoma brzo i jednostavno iz svakog skupa možemo ukloniti osobu za koju znamo da nije zvezda i na taj način smanjiti dimenziju problema. Kada u skupu ostane samo jedna osoba, ona je jedini kandidat da bude zvezda polaznog skupa. Za tu jedinu preostalu osobu onda možemo direktno ispitati da li je zvezda ili nije. Eliminacija ne-zvezde iz skupa se može izvršiti jednostavno. Odaberemo proizvoljne dve osobe u skupu i pitamo se da li osoba A zna osobu B. Ako je odgovor potvrdan, onda osoba A ne može biti zvezda. Ako je odgovor odričan, onda osoba B nije zvezda. Algoritam zasnovan na ovom postupku zadovoljava jednačinu $T(n) = O(1) + T(n-1)$, čije je rešenje $O(n)$. Preostaje još da se proveri da li je preostali jedini kandidat stvarno zvezda. To je moguće uraditi grubom silom za šta nam je dovoljno *2n-1* pitanja, tako da je složenost i ove faze $O(n)$, pa je ukupna složenost $O(n)$.
Biće prikazano jednostavno rešenje koje koristi dva pokazivača. Prvi, i, pokazuje na osobu koja je trenutni kandidat za zvezdu. Drugi, j, pokazuje na osobu za koju se proverava da li je tekući kandidat za zvezdu poznaje. Ukoliko je ne poznaje, i je i dalje kandidat, a j sigurno nije zvezda, pa se j pomera na sledeću osobu. Vreme učitavanja matrice je $O(n^2)$ čime se gubi na efikasnosti.

```
int zvezda(const vector<vector<int>>& poznaje) {
    int i = 0; j = 1;
    while (j < n) { // invarijanta: nijedan element u [0,i) i u (i,j)
        if (poznaje[i][j]) // nije zvezda (0 <= i < j <= n)
            i = j;
        j++;
    }
    if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
        return i;
    return -1;
}
```

3.5 Induktivno-rekurzivna konstrukcija - apsolutni pobednik na glasanju (Bojer-Murov algoritam)

Problem: Održano je glasanje i glasalo se za više kandidata. Osoba je apsolutni pobednik ako je dobila strogo više glasova nego svi ostali kandidati zajedno. Definisati algoritam koji na osnovu niza svih glasačkih listića sa glasanja određuje da li postoji apsolutni pobednik i koji je.
Pretpostavimo da želimo da ispitamo postojanje apsolutnog pobednika u skupu od n glasova. Pitanje je kako problem svesti na problem manje dimenzije. Izbacivanje bilo kog pojedinačnog glasa može da promeni rešenje (jer ako apsolutni pobednik ima za jedan glas više od svih ostalih, nakon izbacivanja tog glasa on više neće biti apsolutni pobednik).
Ključni uvid je da ako izbacimo bilo koja dva različita glasa, onda veći skup ima apsolutnog pobednika samo ako ga ima manji i u pitanju je isti apsolutni pobednik. Dokaz: Ako postoji apsolutni pobednik p u širem skupu od n glasova i on ima m glasova, onda je $m > n/2$. Ako su iz skupa izbačena dva glasa koja nisu za apsolutnog pobednika onda p ima i dalje m glasova, a redukovani skup ima $n-2$ elementa, pa važi $m > n/2 > (n-2)/2$. Ako je izbačen jedan glas za osobu p i jedan koji nije za nju, tada p ima $m-1$ glas, a u skupu ima $n-2$ elementa, pa je $m-1 > (n - 2)/2 = n/2 - 1$. Baza indukcije nastaje kada ne možemo više izbacivati parove različitih elemenata. Prazan skup nema apsolutnog pobednika, a neprazan skup u kome ne postoje dva različita elementa sadrži glasove samo za jednog kandidata koji je očigledno apsolutni pobednik.
Krenućemo od toga da su oba podskupa prazna i polako ćemo određivati njihov sadržaj obrađujući jedan po jedan glas. Dovoljno je samo da pamtimo osobu za koju su svi glasovi iz prvog skupa, kao i broj tih glasova. Ako je prvi skup prazan ili ako je tekući glas za osobu za koju su svi glasovi iz prvog skupa, tekući glas možemo dodati u prvi skup. Ako prvi skup nije prazan i ako je tekući glas za neku drugu osobu od one za koju su svi glasovi iz prvog skupa, onda ćemo taj glas kao i jedan glas za osobu iz prvog skupa prebaciti u drugi skup. Složenost i faze određivanja i faze provere kandidata je očigledno linearna, pa je ceo algoritam složenosti $O(n)$.

```
bool apsolutniPobednik(int glasovi[], int n, int& pobednik) {
    int kandidat;
    int glasovaZaKandidata = 0;
    for (int i = 0; i < n; i++){
        if (glasovaZaKandidata == 0 || glasovi[i] == kandidat) {
            kandidat = glasovi[i];
            glasovaZaKandidata++;
        }
        else
            glasovaZaKandidata--;
    }
    if (glasovaZaKandidata > 0 && count(glasovi, next(glasovi, n),
         kandidat) > n / 2) {
        pobednik = kandidat;
        return true;
    }
    else
        return false;
}
```

3.6 Ojačavanje induktivne hipoteze - izračunavanje vrednosti polinoma

Problem: Dat je niz realnih brojeva a_0, a_1 , ... , a_n i realni broj x. Izračunati vrednost polinoma $P_n(x) = \sum_{i=0}^{n} a_i x^i$.
Jedan način da se problem svede na problem manje dimenzije je da se polinom $P_n(x)$ razloži na zbir $a_n x_n + P_{n-1}(x)$. Izlaz iz rekurzije može predstavljati slučaj $n = 0$ kada je vrednost $P_0(x) = a_0$, a može predstavljati i slučaj $n < 0$ kada je vrednost $P_n(x) = 0$. Da bismo od vrednosti $P_{n-1}(x)$ koju znamo na osnovu induktivne hipoteze mogli da izračunamo $P_n(x)$, moramo da umemo da izračunamo i x^n.
Ako bismo vrednost stepena izračunavali naivno (množenjem sa xn puta), dobili bismo kôd složenosti $O(n^2)$.
Primetimo da vrednost x^n možemo izračunati inkrementalno, ne računajući stepen iz početka, već na osnovu prethodno izračunate vrednosti x^{n-1}. Time ojačavamo induktivnu hipotezu i gradimo funkciju za koju ne pretpostavljamo samo da ume da izračuna vrednost $P_n(x)$ nego ujedno i vrednost x^{n+1}. Invarijanta: v sadrži vrednost $P_{i-1}(x)$, a s vrednost x_i i $0 \leq i \leq n + 1$.

```
void vrednostPolinoma(int a[], int n, int x) {
    int v = 0, s = 1;
    for (int i = 0; i <= n; i++) {
        v += a[i] * s;
        s *= x;
    }
}
```

Prilično je očigledno i da je broj množenja u ovom kodu linearan tj. *O(n)*, pa je ojačavanje induktivne hipoteze pomoglo da se dođe do efikasnijeg algoritma. Polinom $P_n(x)$ možemo zapisati kao $(a_n x^{n-1} + a_{n-1}x^{n-2} + a_1)x + a_0$. Dakle, prvo određujemo vrednost polinoma stepena *n-1* čiji su koeficijenti svi osim poslednjeg, i onda tu vrednost objedinjavamo sa poslednjim koeficijentom. Ova tehnika poznata je kao *Hornerova šema*. Invarijanta petlje je da je za $-1 \leq i \leq n$ vrednost promenljive v jednaka $\sum_{k=i+1}^{n} a_k x^{k-i-1}$

```
int v = 0;
for (int i = n; i >= 0; i--)
    v = v*x + a[i];
```

3.7 Ojačavanje induktivne hipoteze - faktori ravnoteže binarnog drveta

Problem: Definišimo u ovom problemu visinu čvora binarnog drveta kao broj čvorova na putanji od tog čvora do njemu najudaljenijeg lista. Faktor ravnoteže čvora v binarnog drveta definišimo kao razliku visina njegovog levog i desnog poddrveta. Definisati funkciju koja za svaki čvor drveta izračunava faktor ravnoteže.
Pretpostavićemo da je drvo predstavljeno sledećom strukturom.

```
struct cvor {
    int faktorRavnoteze;
    int vrednost;
    cvor *levo, *desno;
};
```

Direktna induktivno-rekurzivna konstrukcija u ovom primeru ne daje rezultate jer faktor ravnoteže čvora ne zavisi od faktora ravnoteže levog i desnog poddrveta, već od visine tih drveta. Sa druge strane, visinu drveta veoma jednostavno možemo izračunati na osnovu induktivno-rekurzivne konstrucije. Ova funkcija zadovoljava jednačinu $T(n) = 2 \cdot T(n/2) + O(1)$, čije je rešenje na osnovu master teoreme *O(n)*.

```
int visina(cvor* koren) {
    if (koren == nullptr)
        return 0;
    return max(visina(koren->levo), visina(koren->desno)) + 1;
}
```

Faktor ravnoteže možemo izračunati veoma jednostavno. Pod pretpostavkom da je drvo balansirano, složenost ove funkcije zadovoljava jednačinu $T(n) = 2 \cdot T(n/2) + O(n)$ i njena složenost je *O(nlogn)*. Ako je drvo izdegenerisano u listu, dobija se da vreme izvršavanja zadovoljava jednačinu $T(n) = T(n-1) + O(n)$ čije je rešenje $O(n^2)$.

```
void izracunajFaktoreRavnoteze(cvor* koren) {
    if (koren != nullptr) {
        izracunajFaktoreRavnoteze(koren->levo);
        izracunajFaktoreRavnoteze(koren->desno);
        koren -> faktorRavnoteze = abs(visina(koren->levo) -
                                       visina(koren->desno));
    }
}
```

Algoritam se može poboljšati ako se visina i faktor ravnoteže računaju istovremeno, tj. ako pojačamo induktivnu hipotezu i pretpostavimo da rekurzivnim pozivima možemo da izračunamo i faktore ravnoteže i visinu poddrveta. Pod pretpostavkom da je drvo balansirano, složenost ovog algoritma zadovoljava jednačinu $T(n) = 2 \cdot T(n/2) + O(1)$, čije je rešenje *O(n)*. Čak i kada drvo nije balansirano, složenost je linearna.

```
int izracunajFaktoreRavnoteze(cvor* koren) {
    if (koren == nullptr)
        return 0;
    int visina_levo = izracunajFaktoreRavnoteze(koren->levo);
    int visina_desno = izracunajFaktoreRavnoteze(koren->desno);
    koren->faktorRavnoteze = abs(visina_levo - visina_desno);
    return max(visina_levo, visina_desno) + 1;
}
```

3.8 Ojačavanje induktivne hipoteze - dijametar binarnog drveta

Problem: Rastojanje između dva čvora binarnog drveta je broj grana na jedinstvenom putu koji ih povezuje. Dijametar drveta je najveće moguće rastojanje dva njegova čvora. Konstruisati efikasan algoritam za određivanje dijametra datog drveta.
Najduži put između dva čvora ili prolazi ili ne prolazi kroz koren. Ako ne prolazi, onda se oba čvora nalaze ili u levom ili u desnom poddrvetu, pa se rastojanje između njih može odrediti na osnovu induktivne hipoteze. Da bismo odredili najduži put koji prolazi kroz koreni čvor, potrebno je da znamo visinu levog i desnog poddrveta. Složenost funkcije *visina* je *O(n)*, dok je složenost izračunavanja dijametra *O(nlogn)* u slučaju balansiranog stabla tj. $O(n^2)$ u opštem slučaju.

```
int visina(cvor* koren) {
    if (koren == nullptr)
        return 0;
    return max(visina(koren->levo), visina(koren->desno)) + 1;
}
```

```
int dijametar(cvor* koren) {
    if (koren == nullptr)
        return 0;
    int dijametar_l = dijametar(koren->levo);
    int dijametar_d = dijametar(koren->desno);
    int dijametar_c = visina(koren->levo) + 2 + visina(koren->desno);
    return max({dijametar_l, dijametar_c, dijametar_d});
}
```

Ponovo možemo ojačati induktivnu hipotezu i visinu izračunavati paralelno sa dijametrom. U ovom slučaju složenost najgoreg slučaja je *O(n)*, čak i kada drvo nije balansirano.

```
void visina_i_dijametar(cvor* koren, int& visina, int& dijametar) {
    if (koren == nullptr) {
        visina = 0;
        dijametar = 0;
        return;
    }
    int visina_l, dijametar_l;
    visina_i_dijametar(koren->levo, visina_l, dijametar_l);
    int visina_d, dijametar_d;
    visina_i_dijametar(koren->desno, visina_d, dijametar_d);
    int dijametar_c = visina_l + 2 + visina_d;
    visina = max(visina_l, visina_d) + 1;
    dijametar = max({dijametar_l, dijametar_c, dijametar_d});
}
```

3.9 Ojačavanje induktivne hipoteze - maksimalni zbir segmenta, Kadanov algoritam

Problem: Definisati efikasnu funkciju koja pronalazi najveći mogući zbir segmenta (podniza uzastopnih elemenata) datog niza brojeva.
Za prazan niz, jedini segment je prazan i njegov je zbir nula. Smatramo da umemo da problem rešimo za proizvoljan niz dužine n i na osnovu toga pokušavamo da rešimo zadatak za niz dužine $n+1$. Segment najvećeg zbira u proširenom nizu se ili ceo sadrži u polaznom nizu dužine n ili čini sufiks proširenog niza, tj. završava se na poslednjoj poziciji. Na osnovu induktivne hipoteze znamo da izračunamo najveći zbir segmenta. Jedan način je da prilikom svakog proširenja niza iznova analiziramo sve segmente koji se završavaju na tekućoj poziciji, ali čak iako to radimo inkrementalno najviše što možemo dobiti je algoritam kvadratne složenosti. Ključni uvid je to da najveći zbir sufiksa koji se završava na tekućoj poziciji možemo inkrementalno izračunati znajući najveći zbir segmenta koji se završava na prethodnoj poziciji. Ako je zbir najvećeg segmenta koji se završava na prethodnoj poziciji i tekućeg elementa pozitivan, onda je to upravo najveći zbir sufiksa proširenog niza. Ako je zbir najvećeg

sufiksa pre proširenja niza i tekućeg elementa negativan, onda je optimalan zbir sufiksa nakon proširenja niza 0 (uzimamo prazan segment). Dakle, proširićemo induktivnu hipotezu i pretpostavićemo da za niz umemo da izračunamo najveći zbir segmenta, ali i najveći zbir sufiksa. Invarijanta je da u svakom koraku petlje znamo ove dve vrednosti (maksimum segmenta i maksimum sufiksa). Složenost ovog algoritma je $O(n)$.

```
int maxSufiks = 0, maxSegment = maxSufiks;
for (int i = 0; i < n; i++) {
    maxSufiks += a[i];
    if (maxSufiks < 0)
        maxSufiks = 0;
    if (maxSegment < maxSufiks)
        maxSegment = maxSufiks;
}
cout << maxSegment << endl;
```

3.10 Ojačavanje induktivne hipoteze - maksimalna suma nesusednih elemenata

Problem: Napiši program koji određuje najveći zbir podniza datog niza nenegativnih brojeva koji ne sadrži dva uzastopna člana niza.
Niz elemenata možemo razložiti na poslednji element i prefiks bez njega. Maksimalni zbir je veći od dva zbira: prvog koji se dobija tako što se poslednji element doda na maksimalni zbir elemenata prefiksa koji ne uključuje pretposlednji element i drugog koji se dobija kao zbir elemenata prefiksa koji uključuje pretposlednji element. Ojačavamo induktivnu hipotezu i pretpostavljamo da za svaki prefiks niza umemo da odredimo upravo te dve vrednosti. Induktivnu hipotezu proširujemo tako što prilikom dodavanja novog elementa maksimalni zbir tekućeg prefiksa sa tim novim elementom određujemo kao zbir tekućeg elementa i zbira prethodnog prefiksa bez tog elementa, dok maksimalni zbir tekućeg prefiksa bez tog novog elementa određujemo kao veći od maksimalnog zbira prethodnog prefiksa sa njegovim poslednjim i maksimalnog zbira prethodnog prefiksa bez njegovog poslednjeg elementa. Kada se petlja završi, veći od dva maksimalna zbira predstavlja traženi globalni maksimum.

```
int x; // O(n)
cin >> x;
int maks_zbir_bez = 0, maks_zbir_sa = x;
for (int i = 1; i < n; i++) {
    int x;        cin >> x;
    int novi_maks_zbir_bez = max(maks_zbir_sa, maks_zbir_bez);
    maks_zbir_sa = maks_zbir_bez + x;
    maks_zbir_bez = novi_maks_zbir_bez;
}
cout << max(maks_zbir_bez, maks_zbir_sa) << endl;
```

3.11 Ojačavanje induktivne hipoteze - broj rastućih segmenata

Problem: Dat je niz a celih brojeva. Definisati efikasan algoritam kojim se određuje koliko u tom nizu postoji rastućih segmenata.
Problem možemo rešiti efikasnije ako promenimo redosled obilaska i umesto fiksiranja levog kraja, fiksiramo desni kraj segmenata. Dakle, želimo da pronađemo broj rastućih segmenata koji se završavaju na poziciji 1, zatim koji se završavaju na poziciji 2itd., sve do pozicije *n-1*. Ključni uvid je da broj segmenata koji se završavaju na poziciji j možemo odrediti veoma jednostavno inkrementalno, znajući broj segmenata koji se završavaju na poziciji *j-1*. Naime, ako je $a_j \leq a_{j-1}$ na poziciji j se ne završava ni jedan rastući segment. U suprotnom, svaki rastući segment koji se završavao na poziciji a_{j-1} može biti produžen elementom a_j, a rastući je i dvočlani segment $[a_{j-1}, a_j]$. Ojačavamo induktivnu hipotezu pretpostavljajući da za svaku poziciju j pored broja svih rastućih segmenata među elementima niza zaključno sa pozicijom j umemo da izračunamo i broj rastućih segmenata koji se završavaju na poziciji j. Složenost ovog postupka je *O(n)*.

```
int ukupanBrojRastucih = 0;
int brojRastucih = 0;
for (int i = 1; i < n; i++) {
    if (a[i] > a[i-1]) {
        brojRastucih++;
        ukupanBrojRastucih += brojRastucih;
    }
    else
        brojRastucih = 0;
}
```

4 Strukture podataka

4.1 Strukture podataka: skalarni tipovi, parovi, torke, slogovi

Da bi algoritmi mogli efikasno funkcionisati potrebno je da podaci koji se obrađuju budu organizovani na način koji omogućava da im se efikasno pristupa i da se efikasno modifikuju i ažuriraju. Za to se koriste *strukture podataka*. Strukture podataka su obično kolekcije podataka koje omogućavaju neke karakteristične operacije. U zavisnosti od implementacije samih struktura izvođenje tih operacija može biti manje ili više efikasno. Prilikom dizajna algoritma često se fokusiramo upravo na operacije koje nam određena struktura nudi, podrazumevajući da će sama struktura podataka biti implementirana na što efikasniji način. Sa stanovišta dizajna algoritma nam je potrebno samo da znamo koje operacije podrazumeva određena struktura podataka i da imamo neku procenu izvođenja te operacije. U tom slučaju kažemo da algoritme konstruišemo u odnosu na *apstraktne strukture podataka*, čime se prilikom dizajna i implementacije algoritama oslobađamo potrebe da brinemo o detaljima implementacije same strukture podataka.
Pojedinačni podaci se čuvaju u promenljivama osnovnih, *skalarnih tipova podataka* (to su obično celobrojni i realni tipovi podataka, karakterski tip, logički tip, pa čak i niske, ako se gledaju kao atomički podaci, bez analize njihovih delova).
U jeziku C++ se tip *para* navodi kao **pair<T1, T2>** gde su T1 i T2 tipovi prve i druge komponente. Pristup prvom elementu para vrši se poljem *first*, a drugom poljem *second*. Par se od vrednosti dobija funkcijom *make_pair*, a moguća je inicijalizacija korišćenjem vitičastih zagrada. Funkcijom *tie* moguće je par razložiti na komponente.
U jeziku C++ se tip *torke* navodi kao **tuple<T0, T1, ...>** gde su Ti redom tipovi komponenata torke. Pristup *i*-tom elementu torke vrši se funkcijom *get<i>*. Torka se od pojedinačnih vrednosti gradi funkcijom *make_tuple*. Funkcijom *tie* moguće je torku razložiti na komponente. Pod pretpostavkom da se pojedinačne komponente mogu porediti i torke se mogu porediti relacijskim operatorima.
Nema velike razlike između definisanja i korišćenja struktura (*slogova*) u jeziku C i jeziku C++. Svakom pojedinačnom podatku pristupa se na osnovu naziva. Prilikom korišćenja slogova potrebno je eksplicitno definisati novi tip podataka, pa njihovo korišćenje zahteva malo više programiranja, nego korišćenje parova i torki.

4.2 Nizovi: statički, dinamički, višedimenzionalni

Nizovi predstavljaju praktično osnovne kolekcije podataka. Osnovna karakteristika nizova je to da omogućavaju efikasan pristup (u složenosti O(1)) elementu niza na osnovu njegove pozicije (indeksa). Pristup van granica niza obično prouzrokuje grešku u programu. U zavisnosti od toga da li je broj elemenata niza poznat (i ograničen) u trenutku pisanja i prevođenja programa ili se određuje i menja tokom izvršavanja programa nizove delimo na statičke i dinamičke. Osnovna operacija u radu sa nizovima je pristup i-tom elementu.
Rad sa *statičkim* nizovima u jeziku C++ je veoma sličan radu sa statičkim nizovima u C-u.
Kada nije unapred poznata dimenzija niza ili kada se očekuje da će se ona prilično menjati pri raznim pokretanjima programa poželjno je korišćenje *dinamičkih* nizova. Dinamički nizovi obično podržavaju rezervaciju prostora za određeni broj elemenata. Takođe, dopuštena operacija je dodavanje na kraj niza. U slučajevima kada u nizu nema dovoljno prostora za smeštanje elemenata vrši se automatska realokacija. U jeziku C++ dinamički nizovi su podržani klasom **vector<T>** gde je T tip podataka koji se smeštaju u vektor. Prilikom konstrukcije moguće je navesti inicijalni broj elemenata. Metod *size* se može koristiti za određivanje veličine niza. Metodom *resize* vrši se efektivna promena veličine niza. Ako je veličina manja, krajnji elementi će biti obrisani. Ako je veličina veća (i ako ima dovoljno memorije) novi elementi niza će biti inicijalizovani na svoju podrazumevanu vrednost (za int to je vrednost 0). Moguće je pored veličine niza i eksplicitno navesti vrednost na koju želimo da elementi niza budu inicijalizovani. Rezervaciju prostora (ali bez promene veličine niza) moguće je postići pozivom metode *reserve*. Metodom *push_back* vrši se dodavanje elementa na kraj niza (time se veličina niza povećava za jedan).
U nekim slučajevima su nam potrebni *višedimenzionalni* nizovi. U dvodimenzionom slučaju jedno od osnovnih pitanja je da li su u pitanju matrice kod kojih sve vrste imaju isti broj elemenata ili su u pitanju nizovi vrsta kod kojih svaka vrsta može imati različit broj elemenata. I višedimenzionalni nizovi mogu biti statički i dinamički. U jeziku C++ statički višedimenzionalni nizovi se koriste na veoma sličan način kao jednodimenzionalni (i veoma slično kao u C-u). Ukoliko dimenzija nije unapred poznata, najlakše nam je da za smeštanje matrice upotrebimo vektor vektora.

4.3 Stek - definicija, implementacija

Stek predstavlja kolekciju podataka u koju se podaci dodaju po LIFO principu - element se može dodati i skinuti samo na vrha steka. U jeziku C++ stek se realizuje klasom **stack<T>** gde T predstavlja tip elemenata na steku. Metode:

- *push* - postavlja dati element na vrh steka
- *pop* - skida element sa vrha steka
- *top* - očitava element na vrhu steka (pod pretpostavkom da stek nije prazan)
- *empty* - proverava da li je stek prazan
- *size* - vraća broj elemenata na steku

Stek u jeziku C++ je zapravo samo adapter oko neke kolekcije podataka (podrazumeano vektora) koji korisnika tera da poštuje pravila pristupa steku i sprečava da napravi operaciju koja nad stekom nije dopuštena (poput pristupa nekom elementu ispod vrha).

4.4 Stekovi - upotreba prilikom eliminacija rekurzije (QuickSort)

Problem: Implementiraj brzo sortiranje nerekurzivno.
Na steku ćemo čuvati argumente rekurzivnih poziva funkcije sortiranja. Na početku je to par indeksa (0, n-1). Glavna petlja se izvršava sve dok se stek ne isprazni i u njoj se obrađuje par indeksa koji se skida sa vrha steka. Umesto rekukrzivnih poziva njihove ćemo argumente postavljati na vrh steka i čekati da oni budu obrađeni u nekoj od narednih iteracija petlje. Primetimo da se argumenti drugog rekurzivnog poziva obrađuju tek kada se u potpunosti reši potproblem koji odgovara prvom rekurzivnom pozivu, što odgovara ponašanju funkcije kada je zaista implementirana rekurzivno.

```
vector<int> a{3, 5, 4, 2, 6, 1, 9, 8, 7};
stack<pair<int, int>> sortirati;
sortirati.push(make_pair(0, a.size() - 1));

while (!sortirati.empty()) {
    auto p = sortirati.top();
    int l = p.first, d = p.second;
    sortirati.pop();
    if (d - l < 1)
        continue;
    int k = l;
    for (int i = l+1; i <= d; i++)
        if (a[i] < a[l])
            swap(a[++k], a[i]);
    swap(a[k], a[l]);
    sortirati.push(make_pair(l, k-1));
    sortirati.push(make_pair(k+1, d));
}

for (int x : a)
    cout << x << endl;
```

4.5 Stekovi - nerekurzivni DFS

Problem: Implementirati nerekurzivnu funkciju koja vrši DFS oblilazak drveta ili grafa (zadatog pomoću lista suseda).

```
vector<vector<int>> susedi {
    {1, 2}, {3, 4}, {5}, {}, {6, 7}, {8}, {}, {}, {}};

void dfs(int cvor) {
    int brojCvorova = susedi.size();
    vector<bool> posecen(brojCvorova, false);
    stack<int> s;
    s.push(cvor);

    while (!s.empty()) {
        cvor = s.top();
        s.pop();
        if (!posecen[cvor]) {
            posecen[cvor] = true;
            cout << cvor << endl;
            for (int sused : susedi[cvor])
                if (!posecen[sused])
                    s.push(sused);
        }
    }
}
```

4.6 Stekovi - izračunavanje vrednosti postfiksnog izraza

Problem: Napisati program koji izračunava vrednost ispravnog postfiksno zadatog izraza koji sadrži samo jednocifrene brojeve i operatore + i *. Na primer, za izraz 34+5* program treba da izračuna vrednost 35.
Velika prednost postfiksno zapisanih izraza je to što im se vrednost veoma jednostavno izračunava uz pomoć steka. Kada naiđemo na broj postavljamo ga na vrh steka. Kada naiđemo na operator, skidamo dve vrednosti sa vrha steka, primenjujemo na njih odgovarajuću operaciju i rezultat postavljamo na vrh steka. Vrednost celog izraza se na kraju nalazi na vrhu steka.

```
bool jeOperator(char c) {
    return c == '+' || c == '*';
}

int primeniOperator(char op, int op1, int op2) {
    int v;
    switch(op) {
        case '+': v = op1 + op2;
                  break;
        case '*': v = op1 * op2;
                  break;
    }
    return v;
}
```

```
int main() {
    string izraz;
    cin >> izraz;
    stack<int> st;
    for (char c : izraz) {
        if (isdigit(c))
            st.push(c - '0');
        else if (jeOperator(c)) {
            int op2 = st.top();
            st.pop();
            int op1 = st.top();
            st.pop();
            st.push(primeniOperator(c, op1, op2));
        }
    }
    cout << st.top() << endl;
    return 0;
}
```

4.7 Stekovi - prevođenje izraza u postfiksni oblik (Dejkstrin algoritam)

Problem: Napisati program koji vrši prevođenje ispravnog infiksno zadatog izraza u njemu ekvivalentan postfiksno zadati izraz. Pretpostaviti da su svi brojevi jednocifreni. Na primer, za izraz 2*3+4*(5+6) program treba da ispiše 23*456+*+.

Mora se obraćati pažnja na prioritet i asocijativnost operatora. Ključna dilema je šta raditi u situaciji kada se pročita *op2* u izrazu oblika *i1 op1 i2 op2 i3* gde su *i1*, *i2* i *i3* tri izraza, a *op1* i *op2* dva operatora. U tom trenutku na izlazu će se nalaziti izraz *i1* preveden u postfiksni oblik i iza njega izraz *i2* preveden u postfiksni oblik, dok će se operator *op1* nalaziti na vrhu steka operatora. Ukoliko *op1* ima veći prioritet od operatora *op2* ili ukoliko im je prioritet isti, ali je asocijativnost leva, tada je potrebno prvo izračunavati izraz *i1 op1 i2* time što se operator *op1* sa vrha steka prebaci na izlaz. U suprotnom operator *op1* ostaje na steku i iznad njega se postavlja operator *op2*.

Ovo je jedan od mnogih algoritama koje je izveo Edsger Dejkstra i naziva se *Shunting yard algorithm*, što bi se moglo slobodno prevesti kao algoritam sortiranja železničkih vagona.

```
int prioritet(char c) {
    if (c == '+' || c == '-')
        return 1;
    if (c == '*' || c == '/')
        return 2;
    throw "Nije operator";
}
```

```
void prevedi(const string& izraz) {
    stack<char> operatori;
    for (char c : izraz) {
        if (isdigit(c))
            cout << c;
        if (c == '(')
            operatori.push(c);
        if (c == ')') {
            while (operatori.top() != '(') {
                cout << operatori.top();
                operatori.pop();
            }
            operatori.pop();
        }
        if (jeOperator(c)) {
            while (!operatori.empty() && jeOperator(operatori.top())
                && prioritet(operatori.top()) >= prioritet(c)) {
                cout << operatori.top();
                operatori.pop();
            }
            operatori.push(c);
        }
    }
    while (!operatori.empty()) {
        cout << operatori.top();
        operatori.pop();
    }
    cout << endl;
    return 0;
}
```

4.8 Stekovi - izračunavanje vrednosti infiksnog izraza

Problem: Napisati program koji izračunava vrednost ispravno zadatog infiksno zapisanog izraza koji sadrži operatore + i *. Jednostavnosti radi pretpostaviti da su svi operandi jednocifreni brojevi. Na primer, za izraz (3+4)*5+6 program treba da ispiše 41.

```
void primeni(stack<char>& operatori, stack<int>& vrednosti) {
    char op = operatori.top(); operatori.pop();
    int op2 = vrednosti.top(); vrednosti.pop();
    int op1 = vrednosti.top(); vrednosti.pop();
    int v;
    if (op == '+') v = op1 + op2;
    if (op == '*') v = op1 * op2;
    vrednosti.push(v);
}
```

```
int vrednost(const string& izraz) {
    stack<int> vrednosti;
    stack<char> operatori;
    for (char c : izraz) {
        if (isdigit(c))
            vrednosti.push(c - '0');
        else if (c == '('
            operatori.push('(');
        else if (c == ')')
            while (operatori.top() != '('
                primeni(operatori, vrednosti);
        else if (jeOperator(c)) {
            while (!operatori.empty() && jeOperator(operatori.top())
                && prioritet(operatori.top()) >= prioritet(c))
                primeni(operatori, vrednosti);
            operatori.push(c);
        }
    }
    while (!operatori.empty())
        primeni(operatori, vrednosti);
    return vrednosti.top();
}
```

4.9 Stekovi - najbliži veći prethodnik, najbliži veći sledbenik

Najbliži veći prethodnik

Problem: Za svaku poziciju i u nizu celih brojeva a pronaći poziciju $j < i$, takvu da je $a_j > a_i$ i da je j najbliža poziciji i. Za svaku pronađenu poziciju j ispisati element a_j, a ako takva pozicija j ne postoji ispisati karakter -. Na primer, za niz 3, 7, 4, 2, 6, 5, ispisati -, -, 7, 4, 7, 6.

Naivno rešenje zasnovano na linearnoj pretrazi u kom bi se za svaki element redom unazad tražio njemu najbliži veći prethodnik bi bilo veoma neefikasno ($O(n^2)$) i taj najgori slučaj bi se javljao kod neopadajućih nizova.

Pokušajmo da efikasniji algoritam konstruišemo induktivno-rekurzivnom konstrukcijom. Bazu čini jednočlan niz i sigurni smo da početni element nema prethodnika većeg od sebe. Pretpostavimo da za svaki element niza dužine k umemo da odredimo najbližeg većeg prethodnika i razmotrimo kako bismo odredili najbližeg većeg prethodnika poslednjeg elementa u nizu. Analizu krećemo od direktnog prethodnika tekućeg elementa. Ako je taj element strogo veći od tekućeg, on mu je najbliži veći prethodnik. U suprotnom rekurzivno određujemo njegovog najbližeg većeg prethodnika i tako dobijeni element upoređujemo sa tekućim elementom. Taj postupak ponavljamo sve dok ne dođemo do elementa koji je veći od tekućeg elementa ili do situacije u kojoj neki element manji ili jednak od tekućeg nema većih prethodnika.

Pošto je niz kandidata opadajući, kandidati koji bivaju eliminisani se mogu nalaziti samo na kraju niza potencijalnih kandidata. Ovo ukazuje na to da se tekući kandidati za najbližeg većeg prethodnika mogu čuvati na steku i da nije potrebno istovremeno čuvati čitav niz *najblizi_veci_prethodnik*, već samo one kandidate koji nisu eliminisani jer su zaklonjeni nekim većim ili jednakim elementom. U trenutku kada se obrađuje element na poziciji j na steku se nalaze vrednosti ili pozicije i takve da je a_i maksimum elemenata niza a na pozicijama iz intervala [i, j). Složenost najgoreg slučaja je $O(n)$.

```
int n;
cin >> n;
stack<int> s;
for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    while (!s.empty() && s.top() <= x)
        s.pop();
    if (s.empty())
        cout << "-" << endl;
    else
        cout << s.top() << endl;
    s.push(x);
}
```

Najbliži veći sledbenik

Problem: Za svaku poziciju i u nizu celih brojeva odrediti i ispisati poziciju njemu najbližeg sledbenika koji je strogo veći od njega, tj. najmanju od svih pozicija $j > i$ za koje važi $a_i < a_j$. Ako takva pozicija ne postoji, ispisati broj članova niza n. Pozicije se broje od nule. Na primer, za niz 1 3 2 5 3 4 7 5 treba ispisati 1 3 3 6 5 6 8.

Pokazaćemo direktno rešenje. Invarijanta petlje će biti to da se na steku (u rastućem redosledu) nalaze pozicije svih elemenata čiji najbliži veći sledbenik još nije određen. Elementi čije su pozicije na steku će biti uvek u nerastućem redosledu. Za svaki element koji obrađujemo, sa vrha steka skidamo pozicije onih elemenata koji su strogo manji od njega i beležimo da je traženi najbliži veći sledbenik za elemente na tim pozicijama upravo element koji trenutno obrađujemo. Za elemente sa steka koji su veći ili jednaki od tekućeg znamo da nisu u delu niza pre tekućeg elementa imali veće sledbenike, a pošto su oni veći ili jednaki od tekućeg elementa njima ni on nije veći sledbenik, tako da oni ostaju na steku, a na vrh steka se postavlja pozicija tekućeg elementa, jer ni njemu još nismo pronašli većeg sledbenika. Nakon obrade celog niza na steku su ostali elementi koji nemaju većeg sledbenika. Pošto pozicije sledbenika ne saznajemo u redosledu u kojem je potrebno ispisati ih, moramo ih pamtiti u pomoćni niz koji ćemo na kraju ispisati.

```
int n;
cin >> n;
vector<int> a(n);
for (int i = 0; i < n; i++)
    cin >> a[i];
vector<int> p(n);
stack<int> s;

for (int i = 0; i < n; i++) {
    while (!s.empty() && a[s.top()] < a[i]) {
        p[s.top()] = i;
        s.pop();
    }
    s.push(i);
}
while (!s.empty()) {
    p[s.top()] = n;
    s.pop();
}
for (int i = 0; i < n; i++)
    cout << p[i] << endl;
```

4.10 Red pomoću dva steka. Stek pomoću dva reda.

Problem: Implementiraj funkcionalnost reda korišćenjem dva steka.

```
stack<int> ulazni, izlazni;

void prebaci() {
    while (!ulazni.empty()) {
        izlazni.push(ulazni.top());
        ulazni.pop();
    }
}
void push(int x) {
    ulazni.push(x);
}
void pop() {
    if (izlazni.empty())
        prebaci();
    izlazni.pop();
}
int top() {
    if (izlazni.empty())
        prebaci();
    return izlazni.top();
}
```

4.11 Red - definicija, oblici, implementacija

Red predstavlja kolekciju podataka u koju se podaci dodaju po FIFO principu - element se dodaje na kraj i skida samo na početka reda. U jeziku C++ red se realizuje klasom **queue<T>** gde T predstavlja tip elemenata u redu. Podržane su sledeće metode:

- *push* - postavlja dati element na kraj reda
- *pop* - skida element sa početka reda
- *front* - očitava element na početku reda
- *back* - očitava element na kraju reda
- *empty* - proverava da li je red prazan
- *size* - vraća broj elemenata u redu

4.12 Redovi - nerekurzivni BFS

Problem: Implementiraj nerekurzivnu funkciju koja vrši BFS obilazak drveta ili grafa.

```
vector<vector<int>> susedi{
    {1, 2}, {3, 4}, {5}, {}, {6, 7}, {8}, {}, {}, {}};

void dfs(int cvor) {
    int brojCvorova = susedi.size();
    vector<bool> posecen(brojCvorova, false);
    queue<int> s;
    s.push(cvor);

    while (!s.empty()) {
        cvor = s.front();
        s.pop();
        if (!posecen[cvor]) {
            posecen[cvor] = true;
            cout << cvor << endl;
            for (int sused : susedi[cvor])
                if (!posecen[sused])
                    s.push(sused);
        }
    }
}
```

4.13 Redovi - maksimalni zbir segmenta dužine k

Problem: Učitava se n brojeva. Napiši program koji određuje segment uzastopnih k elemenata sa najvećim zbirom.
Najjednostavniji način bi bio da se svi elementi učitaju u niz i da se zatim određuju zbirovi segmenata dužine k.
Zbirove možemo računati inkrementalno. Dva susedna segmenta dužine k imaju zajedničke sve elemente osim prvog elementa levog i poslednjeg elementa desnog segmenta. Da bismo izračunali zbir narednog segmenta od zbira prethodnog segmenta treba da oduzmemo njegov početni element i da na zbir dodamo završni element novog segmenta. Nije nam neophodno da čuvamo istovremeno sve elemente već samo elemente tekućeg segmenta dužine k. Pošto se uklanjaju početni elementi segmenta, a segment se proširuje završnim elementima najbolje je elemente čuvati u redu.

```
int main() {
    int n, k;
    cin >> n >> k;
    queue<double> q;
    double zbir = 0.0;

    for (int i = 0; i < k; i++) {
        double x;
        cin >> x;
        q.push(x);
        zbir += x;
    }

    int maxPocetak = 0;
    double maxZbir = zbir;
    for (int i = 1; i <= n-k; i++) {
        double x;
        cin >> x;
        zbir = zbir - q.front() + x;
        q.pop();
        q.push(x);
        if (zbir >= maxZbir) {
            maxZbir = zbir;
            maxPocetak = i;
        }
    }
    cout << maxPocetak << endl;
    return 0;
}
```

4.14 Redovi - maksimalna bijekcija (aktivna lista)

Problem: Dat je konačni skup A i funkcija $f : A \longrightarrow A$. Pronaći maksimalnu kardinalnost skupa $S \subseteq A$, takva da je restrikcija f na S bijekcija.
Da bismo utvrdili da je funkcija bijekcija na nekom konačnom skupu ona mora biti surjekcija na tom skupu i za svaki element skupa mora postojati bar neki element skupa koji se slika u njega. Dakle, ako postoji neki element skupa u koji se ni jedan element ne slika, on ne može biti deo skupa u kom je f bijekcija. Njega možemo ukloniti iz skupa, time redukovati dimenziju problema i do rešenja doći induktivno-rekurzivnom konstrukcijom. Bazu čini slučaj kada se u svaki element skupa slika bar neki element skupa. Ako je u tom slučaju dimenzija skupa n, pošto se u n elemenata skupa neki element slika, a ukupno imamo n originala, na osnovu Dirihleovog principa možemo lako dokazati da svaki original može da se slika u najviše jednu sliku. Utvrdili smo, da važi teorema koja tvrdi da je svaka surjekcija na konačnom skupu ujedno injekcija, pa je i bijekcija.
Što se tiče implementacije, jasno je da je bitno da za svaki element tekućeg skupa odredimo koliko se originala slika u njega (reći ćemo da je to ulazni stepen slike). Ulazne stepene možemo čuvati u jednom nizu (na poziciji i čuvamo koliko se elemenata slika u element i). Niz lako možemo popuniti tako što ga inicijalizujemo na nulu, prođemo kroz sve originale i za njihove slike uvećamo broj originala koji se u te slike slikaju. Prolaskom kroz taj niz možemo identifikovati sve elemente u koje se ni jedan element ne slika. Njih je potrebno da uklonimo iz skupa. Ulazni stepen slike svakog elementa koji uklanjamo moramo umanjiti za jedan. To ne možemo uraditi istovremeno za sve elemente ulaznog stepena nula, već ih možemo ubaciti u red i zatim obrađivati jedan po jedan. Ako nakon smanjenja ulazni stepen nekog elementa postane nula i njega je potrebno ukloniti iz skupa. Da bi on u nekom trenutku bio uklonjen i obrađen, postavićemo ga u red. Dakle, invarijanta našeg programa biće da se u redu nalaze svi elementi čiji je ulazni stepen nula. Kada se red isprazni znaćemo da nema više elemenata čiji je ulazni stepen nula i funkcija će biti bijekcija na skupu elemenata koji nisu uklonjeni iz polaznog skupa. Pošto nas zanima samo njihov broj, održavaćemo samo promenljivu koja čuva kardinalnost tekućeg skupa i prilikom uklanjanja elemenata iz skupa smanjivaćemo je za jedan.

```
int main() {
    int n;
    cin >> n;
    vector<int> f(n);
    for (int i = 0; i < n; i++)
        cin >> f[i];

    vector<int> ulazniStepen(n, 0);
    for (int i = 0; i < n; i++)
        ulazniStepen[f[i]]++;

    queue<int> q;
```

```
    for (int i = 0; i < n; i++)
        if (ulazniStepen[i] == 0)
            q.push(i);

    int brojElemenata = n;
    while (!q.empty()) {
        int i = q.front();
        q.pop();
        brojElemenata--;
        if (--ulazniStepen[f[i]] == 0)
            q.push(f[i]);
    }
    cout << brojElemenata << endl;
    return 0;
}
```

4.15 Red sa dva kraja - definicija, implementacija, primer

Red sa dva kraja je struktura podataka koja kombinuje funkcionalnost steka i reda, jer se elementi mogu i dodavati i skidati sa oba kraja. U jeziku C++ red sa dva kraja se realizuje klasom **deque<T>**, gde je T tip elementa u redu. Red podržava naredne operacije:

- *pushfront* - dodavanje elemenata na početak reda
- *pushback* - dodavanje elemenata na kraj reda
- *popfront* - uklanjanje elemenata sa početka reda
- *popback* - uklanjanje elemenata sa kraja reda
- *front* - očitava element na početku reda
- *back* - očitava element na kraju reda
- *empty* - provera da li je red prazan
- *size* - broj elemenata u redu

Sve ove operacije su konstantne složenosti *O(1)*.
Još jedna klasa koja pruža isti interfejs je **List<T>**, koja je implementirana pomoću dvostruko povezane liste. Osnovna razlika je to što deque<T> ima pristup elementu na osnovu indeksa u konstantnom vremenu.

Primer: U istoriji se čuva najviše n prethodno posećenih veb-sajtova. Naredbom *back* vraćamo se na prethodno posećeni sajt. Napiši program koji simulira rad pregledača. Učitava se jedna po jedna adresa ili naredba *back*. Kada se učita *back* vraćamo se na prethodni sajt i ispisujemo njegovu adresu. Kada nema prethodnog sajta, ispisuje se -.

```
int main() {
    int n;
    cin >> n;
    string linija;
    deque<string> istorija;
```

```
    while (getline(cin, linija)) {
        if (linija == "back") {
            if (!istorija.empty()) {
                cout << istorija.back() << endl;
                istorija.pop_back();
            }
            else
                cout << "-" << endl;
        }
        else {
            if (istorija.size() == n)
                istorija.pop_front();
            istorija.push_back(linija);
        }
    }
    return 0;
}
```

4.16 Red sa dva kraja - maksimumi segmenata dužine k

Problem: Napisati program koji za dati niz određuje maksimume svih njegovih segmenata dužine k. Na primer, ako je $k = 4$ i ako je dat niz 3, 8, 6, 5, 6, 3, segmenti dužine k su 3, 8, 6, 5, zatim 8, 6, 5, 6 i 6, 5, 6, 3 i njihovi maksimumi su redom 8, 8 i 6.
Naivno rešenje u kom bi se svi elementi smestili u niz i u kom bi se iznova tražio maksimum za svaki segment dužine k bilo bi složenosti $O(n^2)$.
Zadatak možemo rešiti i u složenosti $O(n)$. Elemente tekućeg segmenta možemo čuvati u redu dužine k. Prilikom prelaska na svaki naredni segment, red ažuriramo tako što uklanjamo element sa početka reda, a novi element dodamo na kraj. Maksimum ažuriranog reda želimo da računamo inkrementalno, na osnovu maksimuma reda pre ažuriranja. Međutim, to neće teći tako jednostavno. Ako je maksimum segmenta pre ažuriranja njegov prvi element, nakon njegovog uklanjanja gubimo potpuno informaciju o maksimumu preostalih elemenata. Zato moramo ojačati invarijantu i čuvati više informacija. Kada početni element koji je ujedno maksimum segmenta ispadne iz segmenta, moramo znati maksimum preostalih elemenata. Moramo čuvati maksimum celog segmenta, zatim maksimum dela segmenta iza tog maksimuma, zatim maksimum dela segmenta iza tog maksimuma itd. Nazovimo to nizom karakterističnih maksimuma. Uklanjanje početnog elementa segmenta ne menja taj niz, osim u slučaju kada je on jednak maksimumu. U tom slučaju se taj element uklanja sa početka niza. Razmotrimo sada kako se menja niz karakterističnih maksimuma kada se segment proširuje novim završnim elementom. Svi elementi niza karakterističnih maksimuma na desnom kraju koji su strogo manji od novog elementa segmenta se uklanjaju, jer oni više nisu maksimumi dela segmenta iza sebe. Kada se takvi elementi uklone, novi element se dodaje na kraj niza karakterističnih maksimuma, jer je on maksimum jednočlanog segmenta koji sam čini.

```
int main() {
    int k;
    cin >> k;
    int n;
    cin >> n;
    queue<int> segment;
    deque<int> maksimumi;

    for (int i = 0; i < n; i++) {
        if (i >= k) {
            int prvi = segment.front();
            segment.pop();
            if (maksimumi.front() == prvi)
                maksimumi.pop_front();
        }
        int ai;
        cin >> ai;
        segment.push(ai);
        while(!maksimumi.empty() && ai > maksimumi.back())
            maksimumi.pop_back();
        maksimumi.push_back(ai);
        if (i >= k - 1)
            cout << maksimumi.front() << endl;
    }
    return 0;
}
```

4.17 Red sa prioritetom - definicija, implementacija, primer

Red sa prioritetom je vrsta reda u kome elementi imaju na neki način pridružen prioritet, dodaju se u red jedan po jedan, a uvek se iz reda uklanja onaj element koji ima najveći prioritet od svih elemenata u redu. U jeziku C++ red sa prioritetom se realizuje klasom **priority_queue<T>**, gde je T tip elemenata u redu. Red sa prioritetom podržava sledeće metode:

- *push* - dodaje dati element u red
- *pop* - uklanja element sa najvećim prioritetom iz reda
- *top* - očitava element sa najvećim prioritetom
- *empty* - proverava da li je red prazan
- *size* - vraća broj elemenata u redu

Operacije *push* i *pop* su obično složenosti *O(logk)*, gde je k broj elemenata u redu, dok su ostale operacije složenosti *O(1)*.

4.18 Red sa prioritetom - sortiranje pomoću reda sa prioritetom (HeapSort)

Problem: Napisati program koji sortira niz pomoću reda sa prioritetom.
U pitanju je tzv. algoritam sortiranja uz pomoć hipa tj. hip sort. Naziv dolazi od strukture podataka hip (engl. heap) koja se koristi za implementaciju reda sa prioritetom. U pitanju je varijacija algoritma sortiranja selekcijom u kojem se u svakom koraku najmanji element dovodi na početak niza. Određivanje minimuma preostalih elemenata vrši se klasičnim algoritmom određivanja minimuma niza koji je linerne složenosti, što daje ukupnu složenost sortiranja $O(n^2)$.
Algoritam hip-sort koristi činjenicu da je određivanje i uklanjanje najmanjeg elementa iz reda sa prioritetom prilično efikasna operacija (složenosti $O(logk)$, gde je k broj elemenata u redu sa prioritetom). Stoga se sortiranje može realizovati tako što se svi elementi umetnu u red sa prioritetom, iz koga se zatim pronalazi i uklanja jedan po jedan najmanji element.
Memorijska složenost ove implementacije je $O(n)$, jer se u redu čuva svih n elemenata, a vremenska složenost je $O(nlogn)$ jer se n puta izvode operacije složenosti $O(logn)$. Jednačina koja opisuje fazu umetanja, a i fazu izbacivanja je T(n) = T(n-1) + O(logn), a njeno rešenje je $O(nlogn)$.

```
int main() {
    // elementi su u opadajucem redosledu prioriteta
    priority_queue<int, vector<int>, greater<int>> Q;
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int ai;
        cin >> ai;
        Q.push(ai);
    }
    while (!Q.empty()) {
        cout << Q.top() << " ";
        Q.pop();
    }
    return 0;
}
```

4.19 Red sa prioritetom - k najmanjih elemenata

Problem: Napisati program koji omogućava određivanje k najmanjih od n učitanih brojeva unetih sa ulaza.
Jedan način bi bio da se učita niz i da se sortira, ali je to neefikasno. Potrebno je u svakom trenutku da u nekoj strukturi podataka održavamo k najmanjih do tada viđenih elemenata. U prvoj fazi, dok se ne učita prvih k elemenata, svaki novi element samo ubacujemo u strukturu. Nakon toga svaki novi učitani element poredimo sa najvećim elementom u strukturi podataka i ako je manji od njega, taj najveći element izbacujemo, a novi element ubacujemo umesto njega.

Potrebne operacije nam omogućava red sa prioritetom. Memorijska složenost ove implementacije je *O(k)* jer se u redu čuva najviše k elemenata, a vremenska je *O(nlogk)* jer se *O(n)* puta vrše operacije složenosti *O(logk)* (jednačina je T(n) = T(n-1) + O(logk)).

```
int main() {
    int k; cin >> k;
    priority_queue<int> Q;
    int n; cin >> n;
    for (int i = 0; i < n; i++) {
        int x; cin >> x;
        if (Q.size() < k)
            Q.push(x);
        else if (Q.size() == k && x < Q.top()) {
            Q.pop();
            Q.push(x);
        }
    }
    while (!Q.empty()) {
        cout << Q.top() << endl;
        Q.pop();
    }
    return 0;
}
```

4.20 Red sa prioritetom - pronalaženje medijane

Problem: Napisati program koji omogućava operaciju unošenja novog elementa u niz i određivanja medijane do tog trenutka unetih elemenata.
Naivno rešenje bi podrazumevalo da se svi učitani elementi čuvaju u nizu i da se medijana svaki put računa iznova. Najefikasniji način da se medijana izračuna zahteva *O(k)* operacija gde je k dužina niza, pa bi se ovim dobio algoritam složenosti $O(n^2)$. Slično bi bilo i da se elementi održavaju u stalno sortiranom nizu. Tada bi medijana mogla biti izračunata u vremenu *O(1)*, ali bi umetanje elementa na njegovo mesto zahtevalo *O(k)* operacija, pa bi složenost opet bila $O(n^2)$.
Veoma dobar način da se ovaj problem reši je da u svakom trenutku u jednoj (levoj) kolekciji čuvamo sve elemente koji su manji ili jednaki središnjem, a u drugoj (desnoj) sve one koji su veći ili jednaki središnjem, pri uslovu da ako postoji paran broj elemenata, te dve kolekcije treba da sadrže isti broj elemenata, a ako postoji neparan broj elemenata, desna kolekcija može da sadrži jedan element više. Ako ima neparan broj elemenata, tada je medijana jednaka najmanjem elementu desne kolekcije, a u suprotnom je jednaka aritmetičkoj sredini između najvećeg elementa leve i najmanjeg elementa desne kolekcije. Svaki novi element se poredi sa najmanjim elementom desne kolekcije i ako je manji ili jednak njemu ubacuje se u levu kolekciju, a ako je veći od njega, ubacuje

se u desnu kolekciju. Tada se proverava da li se sredina promenila. Ako se desilo da leva kolekcija ima više elemenata od desne, najveći element leve kolekcije treba da prebacimo u desnu. Ako se desilo da u desnoj kolekciji ima dva elementa više nego u levoj, tada najmanji element desne kolekcije prebacujemo u levu. Te kolekcije mogu da budu redovi sa prioritetom u kojima se najmanja tj. najveća vrednost može očitati u konstantnom vremenu, ukloniti u logaritamskom, isto koliko je potrebno i da se umetne novi element.
Ukupno se vrši n dodavanja elemenata, a u svakom koraku se vrše operacije složenosti $O(logk)$, gde je k trenutni broj elemenata umetnutnih u redove. Otud je ukupna složenost $O(nlogn)$.

```
priority_queue<int, vector<int>, greater<int>> veci_od_sredine;
priority_queue<int, vector<int>, less<int>> manji_od_sredine;

double medijana() {
    if (manji_od_sredine.size() == veci_od_sredine.size())
        return (manji_od_sredine.top() + veci_od_sredine.top()) / 2.0;
    else
        return veci_od_sredine.top();
}

void dodaj(int x) {
    if (veci_od_sredine.empty())
        veci_od_sredine.push(x);
    else {
        if (x <= veci_od_sredine.top())
            manji_od_sredine.push(x);
        else
            veci_od_sredine.push(x);
        if (manji_od_sredine.size() > veci_od_sredine.size()) {
            veci_od_sredine.push(manji_od_sredine.top());
            manji_od_sredine.pop();
        }
        else if (veci_od_sredine.size() > manji_od_sredine.size() +
             1) {
            manji_od_sredine.push(veci_od_sredine.top());
            veci_od_sredine.pop();
        }
    }
}
```

4.21 Red sa prioritetom - silueta zgrada

Problem: Duž obale je postavljena koordinatna osa i za svaku zgradu se zna pozicija levog kraja, visina i pozicija desnog kraja. Napisati program koji izračunava siluetu grada. Svaku zgradu predstavljaćemo strukturom koja sadrži levi kraj zgrade a, zatim desni kraj zgrade b i njenu visinu h.

```
struct zgrada {
    int a, b, h;
    zgrada(int a = 0, int b = 0, int h = 0)
        : a(a), b(b), h(h) {
    }
};
```

Silueta je deo-po-deo konstantna funkcija i određena je intervalima konstantnosti (-∞, x_0), [x_0, x_1),..., [x_{n-1}, $+\infty$), određenim tačkama podele $x_0 < x_1 < ... < x_{n-1}$ i vrednostima 0, h_0,..., h_{n-2} i 0 funkcije na svakom od intervala. Podrazumevamo da su krajnje tačke $-\infty$ i $+\infty$ i da su vrednosti na tim intervalima jednake nuli. Deo-po-deo konstantna funkcija se može predstaviti pomoću n tačaka x_0,..., x_{n-1} i *n-1* vrednosti h_0,..., h_{n-2}. Mi ćemo ovakve funkcije predstavljati pomoću n uređenih parova (x_0, h_0),..., (x_{n-2}, h_{n-2}) i (x_{n-1} , 0). Naš algoritam prima niz uređenih trojki koji opisuje pojedinačne zgrade, a vraća niz uređenih parova koji opisuje siluetu. Svaki uređeni par (x_i, h_i) predstavljaćemo strukturom *promena*, a siluetu ćemo predstavljati vektorom *promena*.

```
struct promena {
    int x, h;
    promena(int x = 0, int h = 0)
        : x(x), h(h) {
    }
};
vector<promena> silueta;
```

Efikasno rešenje možemo postići ako zgrade obrađujemo sleva nadesno. Silueta se menja samo u tačkama u kojma počinje ili se završava neka zgrada. Možemo napraviti niz karakterističnih tačaka koji sadrži sve početke i krajeve zgrada i informaciju o tome da li je tekuća tačka početak ili kraj. Kod svake karakteristične tačke potrebno je da odredimo visinu siluete nakon te tačke. Tu visinu ćemo odrediti kao najveću visinu svih zgrada koje počinju levo od te karakteristične tačke a čiji se desni kraj ne nalazi levo od te tačke. Potrebno je da održavamo strukturu podataka u koju ćemo ubacivati zgradu po zgradu kako nailaze, izbacivati zgrade kako prolaze i u svakom trenutku moći efikasno da pronađemo maksimum trenutno ubačenih zgrada. Ako bismo čuvali podatke o zgradama nekako uređene na osnovu njihovih visina, efikasno bismo pronalazili najvišu, ali bi izbacivanje zgrada na osnovu pozicija krajeva bilo komplikovano. Struktura koja nam to omogućava je red sa prioritetom koji je uređen na osnovu visina zgrada. Problem sa takvim redom je to što je izbacivanje zgrade kada se naiđe na njen desni kraj problematično. Ključni trik je da izbacivanje elemenata iz reda odložimo i da u strukturi podataka dopustimo čuvanje podataka koji su po nekom kriterijumu zastareli i koji ne bi više trebalo da se upotrebljavaju. Pretpostavimo da se u redu sa prioritetom nalaze sve zgrade na čiji smo početak do sada naišli. Kada želimo da pronađemo najvišu zgradu od njih koja se još nije završila, možemo da razmotrimo zgradu na vrhu reda. Moguće je da se

ona još nije završila i u tom slučaju ona predstavlja rešenje. U suprotnom, ako se ta zgrada završila, njoj nije više mesto u redu i možemo da je izbacimo iz reda.

```
vector<promena> napraviSiluetu(vector<zgrada>& zgrade) {
    vector<promena> silueta;

    struct PorediPocetak {
        bool operator() (const zgrada& z1, const zgrada& z2) {
            return z1.a < z2.a;
        }
    };

    sort(begin(zgrade), end(zgrade), PorediPocetak());
    vector<pair<int, bool>> tacke(2 * zgrade.size());
    int i = 0;
    for (auto z : zgrade) {
        tacke[i++] = make_pair(z.a, true);
        tacke[i++] = make_pair(z.b, false);
    }

    sort(begin(tacke), end(tacke), [](auto p1, auto p2){
        return p1.first < p2.first;
    });

    struct PorediVisinu {
        bool operator() (const zgrada& z1, const zgrada& z2) {
            return z1.h < z2.h;
        }
    };

    priority_queue<zgrada, vector<zgrada>, PorediVisinu> pq;
    int z = 0;
    for (auto p : tacke) {
        int x = p.first;
        int je_pocetak = p.second;
        if (je_pocetak)
            while (z < zgrade.size() && zgrade[z].a == x)
                pq.push(zgrade[z++]);
        while (!pq.empty() && pq.top().b <= x)
            pq.pop();
        int h = !pq.empty() ? pq.top().h : 0;
        dodajPromenu(silueta, x, h);
    }

    return silueta;
}
```

Ostaje još pitanje kako novu promenu integrisati u postojeću siluetu. Invarijanta koju želimo da nametnemo na siluetu je da su x koordinate svih uzastopnih promena različite i da ne postoje dve uzastopne promene sa istom visinom. Ako poslednja promena u silueti ima istu x koordinatu kao i promena koja se ubacuje, onda se umesto dodavanja nove promene ažurira visina te poslednje promene, ako je to potrebno. Time se može desiti da nakon ažuriranja poslednja i pretposlednja promena imaju istu visinu, pa je u tom slučaju potrebno ukloniti poslednju promenu. Na kraju, ako nova promena ima istu visinu kao i poslednja promena u silueti, nema potrebe da se dodaje. Ovim se invarijanta održava.

```
void dodajPromenu(vector<promena>& silueta, int x, int h) {
    int n = silueta.size();
    if (n > 0) {
        int xb = silueta[n-1].x;
        int hb = silueta[n-1].h;
        if (xb == x) {
            if (h > hb) {
                silueta[n-1].h = h;
                if (n > 1 && silueta[n - 2].h == h)
                    silueta.pop_back();
            }
        }
        else if (hb != h)
            silueta.push_back(promena(x, h));
    }
    else
        silueta.push_back(promena(x, h));
}
```

4.22 Skupovi i mape - definicija, implementacija, primer

Skup je osnovni matematički pojam. U jeziku C++ skup je podržan kroz dve klase: **set<T>** i **unordered_set<T>**, gde je T tip elemenata skupa. Implementacija je različita (prva je zadata na balansiranim binarnim drvetima, a druga na heš tablicama), pa su im vremenske i prostorne karakteristike donekle različite. Skupovi podržavaju sledeće osnovne operacije:

• *insert* - umeće novi element u skup. Kada se koristi *set* složenost je *O(logk)*, gde je k broj elemenata u skupu, a kada se koristi *unordered_set*, složenost najgoreg slučaja je *O(k)*, dok je prosečna složenost *O(1)*.

• *find* - proverava da li skup sadrži dati element i vraća iterator na njega ili *end* ako je odgovor negativan. Složenost najgoreg slučaja ako se koristi *set* je *O(logk)*, a ako se koristi *unordered_set* je *O(k)*, ali je prosečna složenost *O(1)*.

• *erase* - uklanja dati element iz skupa. Složenost je ista kao u prethodnom slučaju.

Mape su strukture podataka koje skup ključeva iz nekog konačnog domena preslikavaju u neki skup vrednosti. Osnovne operacije su pridruživanje vrednosti datom ključu, očitavanje vrednosti pridružene nekom datom ključu, ispitivanje da li je nekom ključu pridružena vrednost i brisanje elementa sa datim ključem. U jeziku C++ mape su podržane klasama **map<K, V>** i **unordered_map<K, V>**, gde je K tip ključeva, a V tip vrednosti. Slično kao u slučaju skupa, implementacija prve klase je zasnovana na balansiranim binarnim drvetima, a druga na heš tablicama. Na raspolaganju su nam sledeće operacije.

• Osnovni operator u radu sa mapama je operator indeksnog pristupa. Kada se koristi *map*, garantovana složenost ovog operatora je *O(logk)* gde je k broj trenutno pridruženih ključeva u mapi. Kada se koristi *unordered_map* prosečna složenost je *O(1)*, ali konstantni faktor može biti veliki, dok je složenost najgoreg slučaja *O(k)*.

• Metoda *find* vraća iterator koji ukazuje na slog sa datim ključem u mapi. Ako taj ključ ne postoji u mapi, vraća se iterator na kraj mape. Složenost je identična kao u slučaju umetanja.

• Metoda *erase* briše element iz mape. Argument može biti bilo vrednost ključa, bilo iterator koji pokazuje na element koji se uklanja. Složenost brisanja date vrednosti ključa je identična kao u prethodnim slučajevima.

4.23 Skup - eliminacija duplikata

Problem: Napiši program koji određuje da li među učitanih n brojeva ima duplikata.

```
int main() {
    int n;
    cin >> n;
    bool duplikati = false;
    set<int> vidjeni;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        if (vidjeni.find(x) != vidjeni.end()) {
            duplikati = true;
            break;
        }
        duplikati.insert(x);
    }
    cout << (duplikati ? "da" : "ne") << endl;
    return 0;
}
```

Složenost najgoreg slučaja ovog pristupa (a to je slučaj kada nema duplikata) je *O(nlogn)*, jer se n puta izvršava pretraga i umetanje u skup.

4.24 Mapa - brojanje pojavljivanja reči u datoteci

Problem: Napisati program koji izračunava frekvenciju (broj pojavljivanja) svake od reči u tekstu.

```
int main() {
    map<string, int> frekvencije;
    string rec;

    while (cin >> rec)
        frekvencije[rec]++;

    for (auto it : frekvencije)
        cout << it.first << ": " << it.second << endl;
    return 0;
}
```

5 Implementacija struktura podataka

5.1 Dinamički niz - implementacija

S obzirom na to da veoma često broj potrebnih elemenata niza znamo tek u fazi izvršavanja programa, umesto klasičnih, statički alociranih nizova često se koriste *dinamički* alocirani nizovi. Za to se koristi struktura podataka **vector**. Osnovna ideja dinamičkog niza je to da se u startu alocira neki pretpostavljeni broj elemenata i da se, kada se ustanovi da taj broj elemenata nije više dovoljan, izvrši realokacija niza, tako što se alocira novi, veći niz, zatim se u taj novi niz iskopiraju elementi originalnog niza i na kraju se taj stari niz obriše, a pokazivač preusmeri ka novom nizu.
Implementacija osnovne funkcionalnosti dinamičkog niza u jeziku C++ mogla bi da izgleda ovako. Ulogu pokazivača *NULL* iz jezika C ima pokazivač *nullptr*; ulogu funkcije *malloc* ima operator *new*, a ulogu funkcije *free* ima operator *delete*.

```
int* a = nullptr;
int alocirano = 0, int n = 0;

int procitaj(int i) {
    return a[i];
}

void postavi(int i, int x) {
    a[i] = x;
}

void realociraj(int m) {
    int* novo_a = (int*)new int[m];
    alocirano = m;
    if (a != nullptr) {
        copy_n(a, n, novo_a);
        delete[] a;
    }
    a = novo_a;
}

void dodajNaKraj(int x) {
    if (alocirano <= n)
    realociraj(2 * alocirano + 1);
    a[n++] = x;
}

void obrisi() {
    delete[] a;
}
```

5.2 Liste - implementacija

Strukture *liste* podrazumevaju da se podaci čuvaju u memoriji u čvorovima kojima se pored podataka čuvaju pokazivači. U zavisnosti od toga da li se čuvaju samo pokazivači na sledeći ili i na prethodni element razlikuju se: *jednostruko* povezane liste i *dvostruko* povezane liste.
Pod pretpostavkom da su poznati pokazivači na početak i na kraj liste, jednostruko povezane liste dopuštaju dodavanje na početak i na kraj, kao i brisanje sa početka u vremenu *O(1)*, dok brisanje elementa sa kraja zahteva vreme *O(n)*. Umetanje i brisanje elementa iza ili ispred čvora na koji ukazuje poznati pokazivač se može izvršiti u vremenu *O(1)*, međutim pronalaženje pozicije na koju treba ubaciti element obično zahteva prolaz kroz listu i zahteva vreme *O(n)*. Pristup elementu na datoj poziciji zahteva vreme *O(n)*.
Cvorovi su često raštrkani po memoriji, pa su promašaji keš memorije mnogo češći nego u slučaju rada sa strukturama u kojima su podaci u memoriji smešteni povezano. Takođe, zbog čuvanja pokazivača liste zahtevaju mnogo više memorije nego nizovi. Prednosti lista u odnosu na nizove i dekove nastupaju pre svega u situacijama u kojima se u listama čuvaju veliki podaci. Tada se tokom realokacije dinamičkih nizova kopiraju velike količine podataka, što može biti neefikasno, pa je korisnije upotrebiti liste kod kojih se podaci ne realociraju.

5.3 Dvostruko povezana lista - implementacija

Pod pretpostavkom da su poznati pokazivači na početak i na kraj liste, dvostruko povezane liste dopuštaju i dodavanje i brisanje i sa početka i sa krajaliste u vremenu *O(1)*. Ostale operacije se izvršavaju u istom vremenu kao i kod jednostruko povezanih lista. Mane dvostruko povezanih u odnosu na jednostruko povezane liste su to što zbog čuvanja pokazivača na prethodne elemente zahtevaju više memorije i pojedinačne operacije mogu biti malo sporije jer se zahteva ažuriranje više pokazivača. Prednosti su to što omogućuju efikasnije izvršavanje nekih operacija.

5.4 Dek - implementacija

Jedna veoma korisna struktura podataka je red sa dva kraja koja kombinuje funkcionalnost steka i reda. Ako se za implementaciju koriste jednostruko povezane liste, tada se u vremenu *O(1)* može vršiti ubacivanje na početak i brisanje sa početka, kao i ubacivanje na kraj. Brisanje sa kraja je operacija složenosti *O(n)*, čak i kada se čuva pokazivač na poslednji element (jer ne možemo da pronađemo pokazivač na pretposlednji element). Ako se za implementaciju koriste dvostruko povezane liste, tada se ubacivanje i brisanje i sa početka i sa kraja može izvršiti u vremenu *O(1)*. Isto je i sa umetanjem elementa u sredinu (kada se zna njegova pozicija). Međutim, indeksni pristup u najgorem slučaju zahteva vreme *O(n)*.
Jedini mogući način pretrage je linearna pretraga, čak i kada su elementi u redu sortirani.

Dek (engl. deque) je struktura podataka koja se često koristi za implementaciju redova sa dva kraja. Slično kao dvostruko povezane liste, dek omogućava dodavanje i na početak i na kraj u vremenu *O(1)*. Važna prednost u odnosu na dvostruko povezane liste je to što je indeksni pristup moguć u vremenu *O(1)*. Ovim je omogućena i binarna pretraga, što može nekada biti veoma važno. Dodavanje i brisanje elemenata sa sredine nije moguće izvršiti efikasno (te operacije zahtevaju vreme *O(n)*). Dek možemo zamisliti kao niz segmenata iste, fiksne veličine. Svaki segment je struktura koja sadrži niz elemenata (bilo statički, bilo dinamički alociran) koji predstavlja neki deo reda.
Dek čuva niz pokazivača na pojedinačne segmente. Dva segmenta su karakteristična: levi segment u kom se nalazi početak reda i desni segment u kom se nalazi kraj reda. Oba mogu biti samo delimično popunjeni. U svakom segmentu se čuva prva slobodna pozicija na koju se može dodati naredni element. U praznom deku levi i desni segment su susedni i prazni (tekući element levog segmenta je njegov desni kraj, a desnog segmenta je njegov levi kraj). Dodavanje elementa na početak je veoma jednostavno ako levi segment nije popunjen do kraja. Element se samo dodaje na prvu slobodnu poziciju i ona se pomera nalevo. Kada je levi segment potpuno popunjen, prelazi se na popunjavanje prethodnog segmenta, ako on postoji. Ako ne postoji, onda se vrši realokacija deka i proširuje se njegov broj segmenata. Prilikom realokacije vrši se samo proširivanje niza pokazivača na segmente, a ne samih segmenata, što je veoma značajno ako se u deku čuvaju veći objekti. Prilikom realokacije, pokazivači na postojeće segmente se u proširenom nizu smeštaju na sredinu, a levo i desno od njih se smeštaju pokazivači na novo alocirane segmente koji su inicijalno prazni. Realokacijom se dobijaju segmenti levo od tekućeg potpuno popunjenog segmenta i dodavanje na početak se vrši u njih. Dodavanje na desni kraj teče potpuno analogno. Brisanje sa levog kraja se vrši slično (uklanjanjem elemenata iz levog segmenta i prelaskom na naredni segment ako se nakon brisanja levi segment potpuno ispraznio). Brisanje sa desnog kraja je analogno. Indeksni pristup elementu je moguće izvršiti u vremenu *O(1)*, tako što se prvo odredi kom segmentu pripada traženi element, a onda se pročita odgovarajući element iz tog segmenta. Jednostavan trik je da se traženi indeks uveća za prazan broj elemenata na levom kraju levog segmenta. Tada se pozicija segmenta u kom se element nalazi može jednostavno izračunati kao zbir pozicije levog segmenta i celobrojnog količnika indeksa i i veličine jednog segmenta, dok se pozicija unutar tog segmenta određuje kao ostatak u tom deljenju.

```
struct segment {
    int* podaci;
    int popunjeno;
    int tekuci;
};

segment* alocirajSegment(int velicina, int smer) {
    segment* novi = new segment();
    novi->podaci = new int[velicina];
    novi->popunjeno = 0;
```

```
    novi->tekuci = smer == 1 ? 0 : velicina - 1;
    return novi;
}

void obrisiSegment(segment* s) {
    delete[] s->podaci;
    delete s;
}

struct dek {
    segment** segmenti;
    int brojSegmenata;
    int velicinaSegmenata;
    int levo, desno;
};

void realocirajDek(dek& d, int brojSegmenata) {
    segment** noviSegmenti = new segment*[brojSegmenata];
    int uvecanje = (brojSegmenata - d.brojSegmenata) / 2;
    if (d.segmenti != nullptr)
        for (int i = 0; i < d.brojSegmenata; i++)
            noviSegmenti[uvecanje + i] = d.segmenti[i];
    for (int i = 0; i < uvecanje; i++)
        noviSegmenti[i] = alocirajSegment(d.velicinaSegmenata, -1);
    for (int i = uvecanje + d.brojSegmenata; i < brojSegmenata; i++)
        noviSegmenti[i] = alocirajSegment(d.velicinaSegmenata, 1);
    delete[] d.segmenti;
    d.segmenti = noviSegmenti;
    d.brojSegmenata = brojSegmenata;
    d.levo += uvecanje;
    d.desno += uvecanje;
}

int& iti(const dek& d, int i) {
    int uLevom = d.segmenti[d.levo]->popunjeno;
    i += d.velicinaSegmenata - uLevom;
    segment* s = d.segmenti[d.levo + i / d.velicinaSegmenata];
    return s->podaci[i % d.velicinaSegmenata];
}

int dodajNaPocetak(dek& d, int x) {
    if (d.segmenti[d.levo]->popunjeno == d.velicinaSegmenata) {
        if (d.levo == 0)
            realocirajDek(d, d.brojSegmenata * 2);
        d.levo--;
    }
    segment *s = d.segmenti[d.levo];
    s->podaci[s->tekuci--] = x;
    s->popunjeno++;
}
```

```
int dodajNaKraj(dek& d, int x) {
    if (d.segmenti[d.desno]->popunjeno == d.velicinaSegmenata){
        if (d.desno == d.brojSegmenata - 1)
        realocirajDek(d, d.brojSegmenata * 2);
        d.desno++;
    }
    segment *s = d.segmenti[d.desno];
    s->podaci[s->tekuci++] = x;
    s->popunjeno++;
}

void obrisiDek(dek& d) {
    for (int i = 0; i < d.brojSegmenata; i++)
    obrisiSegment(d.segmenti[i]);
    delete[] d.segmenti;
}
```

5.5 Binarno drvo - implementacija

Drveta su strukture podataka koje se koriste za predstavljanje hijerarhijskih odnosa između delova. Razmotrićemo dve posebne organizacije binarnih drveta: uređena binarna drveta (tj. binarna drveta pretrage) koja se koriste za implementaciju skupova, mapa (rečnika), multiskupova i multimapa, kao i hipove koji se koriste za implementaciju redova sa prioritetom.
Binarno drvo je rekurzivno definisani tip podataka: ili je prazno ili sadrži neki podatak i levo i desno poddrvo. Uobičajeni način za predstavljanje drveta u jeziku C++ je preko čvorova uvezanih pomoću pokazivača.

```
struct cvor {
    int x;
    cvor *levo, *desno;
};
```

Pošto je drvo rekurzivno-definisana struktura podataka, najlakše je funkcije koje operišu sa drvetima realizovati rekurzivno. U nekim situacijama je moguće relativno lako eliminisati rekurziju, dok je u nekim drugim situacijama implementiranje nerekurzivnih operacija komplikovano (i zahteva korišćenje steka).

5.6 Binarno drvo pretrage (uređeno binarno drvo) - formiranje, pretraga

Drvo je binarno drvo pretrage ako je prazno ili ako je njegovo levo i desno poddrvo uređeno i ako je čvor u korenu veći od svih čvorova u levom poddrvetu i manji od svih čvorova u desnom poddrvetu. U multiskupovima i multimapama je dozvoljeno postojanje duplikata u drvetu, ali u običnim skupovima i mapama nije.

Nije dovoljno proveriti da je vrednost u svakom čvoru veća od vrednosti u korenu levog poddrveta i vrednosti u korenu desnog poddrveta. Ispravna funkcija zahteva izračunavanje najmanje vrednosti desnog poddrveta i najveće vrednosti levog poddrveta. Ako se njihovo računanje uvek izvršava iz početka, provera će biti neefikasna.

```
bool jeUredjeno(cvor* drvo, int& minV, int& maxV) {
    if (drvo == nullptr) {
        minV = numeric_limits<int>::max();
        maxV = numeric_limits<int>::min();
        return true;
    }
    int minL, maxL;
    bool uredjenoL = jeUredjeno(drvo->levo, minL, maxL);
    if (!uredjenoL)
        return false;
    int minD, maxD;
    bool uredjenoD = jeUredjeno(drvo->desno, minD, maxD);
    if (!uredjenoD) return false;
    if (drvo->x <= maxL) return false;
    if (drvo->x >= minD) return false;
    minV = levo == nullptr ? drvo->x : minL;
    maxV = desno == nullptr ? drvo->x : maxD;
    return true;
}
```

Umetanje u binarno drvo pretrage se uvek vrši na mesto lista. Vrednost koja se umeće se poredi sa vrednošću u korenu, ako je manja od nje umeće se levo, a ako je veća od nje umeće se desno. Ako je jednaka vrednosti u korenu, razlikuju se slučaj u kom se duplikati dopuštaju i slučaj u kom se ne dopuštaju.

```
cvor* napraviCvor(int x) {
    cvor* novi = new cvor();
    novi->levo = novi->desno = nullptr;
    novi->x = x;
    return novi;
}

cvor* ubaci(cvor* drvo, int x) {
    if (drvo == nullptr)
        return napraviCvor(x);
    if (x < drvo->x)
        drvo->levo = ubaci(drvo->levo, x);
    else if (x > drvo->x)
        drvo->desno = ubaci(drvo->desno, x);
    return drvo;
}
```

5.7 Binarno drvo pretrage (uređeno binarno drvo) - brisanje

Nakon završetka rada sa drvetom potrebno ga je ukloniti iz memorije.

```
void obrisi(cvor* drvo) {
    if (drvo != nullptr) {
        obrisi(drvo->levo);
        obrisi(drvo->desno);
        delete drvo;
    }
}
```

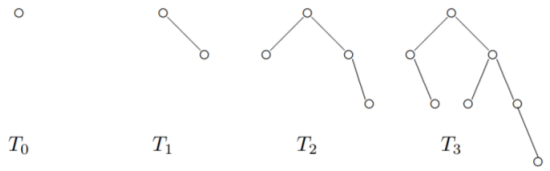
Brisanje iz uređenog binarnog drveta je komplikovanije.

```
cvor* obrisiMin(cvor* drvo, int& x) {
    if(drvo == nullptr)
        return nullptr;
    if(drvo->levo == nullptr) {
        cvor* desno = drvo->desno;
        x = drvo->x;
        delete drvo;
        return desno;
    }
    drvo->levo = obrisiMin(drvo->levo, x);
    return drvo;
}

cvor* obrisi(cvor* drvo, int x) {
    if (drvo == nullptr)
        return nullptr;
    if (x < drvo->x)
        drvo->levo = obrisi(drvo->levo, x);
    else if (x > drvo->x)
        drvo->desno = obrisi(drvo->desno, x);
    else {
        if (drvo->desno == nullptr) {
            cvor* levo = drvo->levo;
            delete drvo;
            return levo;
        }
        else{
            int min;
            drvo->desno = obrisiMin(drvo->desno, min);
            drvo->x = min;
        }
    }
    return drvo;
}
```

5.8 AVL drvo - definicija, složenost, umetanje, rotacije

AVL drveta su struktura podataka koja garantuje da složenost ni jedne od operacija traženja, umetanja i brisanja u najgorem slučaju nije veća od *O(logn)*, gde je n broj elemenata. Posle svake operacije ulaže se dodatni napor da se drvo uravnoteži, tako da visina drveta uvek bude *O(logn)*. Pri tome se uravnoteženost drveta definiše tako da se može lako održavati. Preciznije, AVL drvo se definiše kao uređeno (pretraživačko) binarno drvo kod koga je za svaki čvor apsolutna vrednost razlike visina levog i desnog poddrveta manja ili jednaka od jedan.

Visina h AVL drveta sa n čvorova zadovoljava uslov $h < 2log_2 n$. Dokaz indukcijom po visini drveta. Primetimo da visina drveta ne određuje jednoznačno broj čvorova u drvetu. Za dokazivanje navedene nejednakosti kritična su drveta sa najmanjim brojem čvorova za datu visinu. Stoga se u dokazu bavimo najmanjim drvetima date visine. Neka je T_h AVL drvo visine $h \geq 0$ sa najmanjim mogućim brojem čvorova. Drvo T_0 sadrži samo koren, a drvo T_1 koren i jedno dete, recimo desno. Za $h \geq 2$ drvo T_h može se formirati na sledeći način: njegovo poddrvo manje visine h-2 takođe treba da bude AVL drvo sa minimalnim brojem čvorova, dakle T_{h-2}; slično, njegovo drugo poddrvo treba da bude T_{h-1}. Drveta opisana ovakvom rekurentnom jednačinom zovu se Fibonačijeva drveta.



Označimo sa n_h minimalni broj čvorova AVL drveta visine h, tj. broj čvorova drveta T_h. Važi $n_0 = 1$, $n_1 = 2$, $n_2 = 4$,... Prema definiciji Fibonačijevih drveta važi $n_h = n_{h-1} + n_{h-2} + 1$, za $h \geq 2$. S obzirom na to da je $n_h > n_{h-1}$ za svako $h \geq 2$, važi: $n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} + 1 > 2n_{h-2}$.
Dokažimo indukcijom da važi tvrđenje $n_h > 2^{h/2}$. Za $h = 0$ i $h = 1$ tvrđenje važi. Pretpostavimo da tvrđenje važi za h-2, odnosno da važi $n_{h-2} > 2^{(h-2)/2}$. Na osnovu veze $n_h > 2n_{h-2}$, važi $n_h > 2 \cdot 2^{(h-2)/2} = 2^{h/2-1+1} = 2^{h/2}$, odnosno nakon logaritmovanja obe strane dobijamo $h < 2\log_2 n_h \leq 2\log_2 n$ jer je po pretpostavci za sva drveta visine h broj čvorova n veći ili jednak od n_h. Dakle, visina drveta je *O(logn)* u odnosu na broj čvorova n.
Prilikom umetanja novog elementa u AVL drvo postupa se najpre na način uobičajen za uređeno binarno drvo: pronalazi se mesto čvoru, pa se u drvo dodaje novi list sa ključem jednakim zadatom broju. Čvorovima na putu koji se tom prilikom prelaze odgovaraju razlike visina levog i desnog poddrveta, tzv. faktori ravnoteže iz skupa $\{0, 1, -1\}$. Posebno je interesantan poslednji čvor na tom putu koji ima faktor ravnoteže različit od nule, tzv. kritični čvor. Ispostavlja se da je prilikom umetanja elementa u AVL drvo dovoljno uravnotežiti poddrvo sa korenom u kritičnom čvoru.

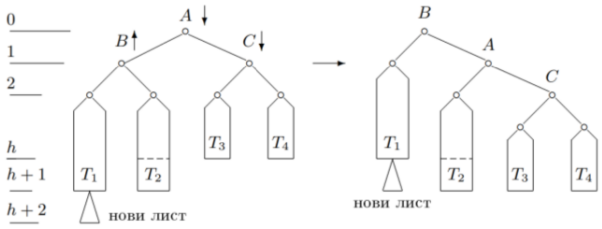
Pretpostavimo da je faktor ravnoteže u kritičnom čvoru jednak 1. Novi čvor N može da završi u:

• desnom poddrvetu – u tom slučaju poddrvo kome je koren kritični čvor ostaje AVL

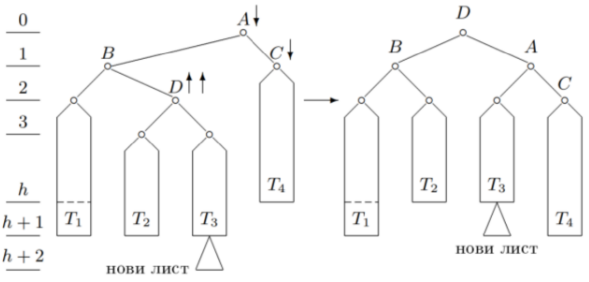
• levom poddrvetu – u tom slučaju drvo prestaje da bude AVL i potrebno je intervenisati. Može se dokazati indukcijom po dužini putanje od kritičnog čvora do lista koristeći činjenicu da je faktor ravnoteže svih njenih čvorova 0. Prema tome da li je novi čvor dodat levom ili desnom poddrvetu levog poddrveta, razlikujemo dva slučaja:

– u slučaju kada je novi čvor dodat levom poddrvetu levog poddrveta na drvo se primenjuje rotacija: koren levog poddrveta B se "podiže" i postaje koren poddrveta kome je koren bio kritičan čvor, a ostatak drveta preuređuje se tako da drvo i dalje ostane uređeno binarno drvo. Drvo T_1 se podiže za jedan nivo ostajući dalje levo poddrvo čvora B; drvo T_2 ostaje na istom nivou, ali umesto desnog poddrveta B postaje levo poddrvo A; desno poddrvo A spušta se za jedan nivo. Pošto je A kritičan čvor, faktor ravnoteže čvora B je 0, pa drveta T_1 i T_2 imaju istu visinu. Drveta T_3 i T_4 ne moraju imati istu visinu, jer čvor C nije na putu od kritičnog čvora do mesta umetanja. Novi koren poddrveta postaje čvor B, sa faktorom ravnoteže 0.

– slučaj kada je novi čvor dodat desnom poddrvetu levog poddrveta je komplikovaniji; tada se drvo može uravnotežiti dvostrukom rotacijom. Novi čvor poddrveta umesto kritičnog čvora postaje desno dete levog deteta kritičnog čvora.



AVL rotacija



AVL dvostruka rotacija

U oba slučaja visina poddrveta kome je koren kritični čvor posle uravnotežavanja ostaje nepromenjena. Uravnotežavanje poddrveta kome je koren kritičan čvor zbog toga ne utiče na ostatak drveta. Uz svaki čvor drveta čuva se njegov faktor ravnoteže, jednak razlici visina njegovog levog i desnog poddrveta; za AVL drvo su te razlike elementi skupa $\{-1, 0, 1\}$. Uravnotežavanje postaje neophodno ako je faktor nekog čvora iz skupa $\{-1, 1\}$, a novi čvor se umeće na pogrešnu stranu. Brisanje iz AVL drveta je komplikovanije, kao i kod običnog uređenog binarnog drveta. U opštem slučaju se uravnotežavanje ne može izvesti pomoću samo jedne ili dve rotacije. Granica potrebna za broj rotacija je *O(logn)*. Svaka rotacija zahteva konstantni broj koraka, pa je vreme izvršavanja brisanja takođe ograničeno odozgo sa *O(logn)*.

5.9 Crveno-crno drvo - definicija, složenost

Crveno-crno drvo (engl. red-black tree, RBT) je uređeno binarno drvo koje dodatno mora da zadovolji sledeće invarijante.

1. Svaki čvor je ili crven ili crn.
2. Koren je crn.
3. Svi listovi su crni i ne sadrže vrednosti (označavamo ih sa NIL).
4. Svi crveni čvorovi imaju tačno dva crna deteta.
5. Sve putanje od nekog čvora do njegovih listova sadrže isti broj crnih čvorova.

Navedena svojstva nam garantuju da će svaka putanja od korena do njemu najdaljeg lista biti najviše duplo duža nego putanja do njegovog najbližeg lista. Zaista, najkraća putanja do lista će se sastojati samo od crnih čvorova, dok će se se u najdužoj putanji naizmenično smenjivati crveni i crni listovi. Ovo svojstvo nam garantuje određeni vid balansiranosti drveta i logaritamsku složenost operacija.

Dokažimo da su ovi uslovi dovoljni da bi se garantovalo da visina drveta logaritamski zavisi od broja čvorova.

Neka je *h(v)* visina poddrveta čiji je koren čvor v tj. broj čvorova od čvora v do najdaljeg lista (ne računajući čvor v) i neka je h_b*(v)* takozvana crna visina poddrveta čiji je koren čvor v tj. broj crnih čvorova od čvora v do bilo kog lista (ne računajući čvor v ako je on crn).

Dokažimo da svako drvo sa korenom u v ima bar $2^{h_b(v)}$-1 unutrašnjih čvorova. Dokaz teče indukcijom po visini drveta tj. po vrednosti *h(v)*.

• Bazu čini slučaj *h(v)* = 0. Visinu nula ima jedino NIL čvor, pa je tada i $h_b(v) = 0$ i $2^{h_b(v)}$-1 = 0, pa je zahtev leme trivijalno ispunjen (drvo ima bar 0 unutrašnjih čvorova).

• Pretpostavimo da svako drvo čija je visina k ima bar $2^{h_b(v)}$-1 čvorova. Neka drvo ima koren u čvoru *v'* i neka je *h(v') = k+1*. Pošto je *h(v')* > 0, čvor *v'* je unutrašnji i ima dva potomka koji su koreni drveta visine k. Za oba potomka važi da im je crna visina ili $h_b(v')$ (ako je potomak crven) ili $h_b(v')$ - 1 (ako je potomak crn). Na osnovu induktivne hipoteze važi da ta dva poddrveta imaju bar $2^{h_b(v)-1}$-1 unutrašnjih čvorova, pa drvo sa korenom u *v'* ima bar $2(2^{h_b(v')-1}$ - 1) + 1 = $2^{h_b(v')}$ - 1 potomaka.

5.10 Hip - definicija, implementacija

Maks-hip (engl. Max-heap) je binarno drvo koje zadovoljava uslov da je svaki nivo osim eventualno poslednjeg potpuno popunjen, kao i da je vrednost svakog čvora veća ili jednaka od vrednosti u njegovoj deci. Omogućava veoma efikasno određivanje maksimalnog elementa u sebi (on se uvek nalazi u korenu), a i veoma efikasno njegovo uklanjanje.
Min-hip se definiše analogno, jedino što se zahteva da je vrednost svakog čvora manja ili jednaka od vrednosti u njegovoj deci.
Pošto je operacija umetanja novog elementa u hip efikasna, ova struktura podataka je veoma dobar kandidat za implementaciju reda sa prioritetom. Ako se koren nalazi na poziciji *i*, onda se njegovo levo dete nalazi na poziciji *2i+1*, a desno dete na poziciji *2i+2*, dok mu se roditelj nalazi na poziciji $[\frac{i-1}{2}]$.

```
int roditelj(int i) {
    return (i-1) / 2;
}
int levoDete(int i) {
    return 2*i + 1;
}
int desnoDete(int i) {
    return 2*i + 2;
}
```

```
int najveci(const vector<int>& hip) { // O(1)
    return hip[0]; // uvek je max u korenu
}
```

Kako bismo mogli realizovati operaciju uklanjanja najvećeg elementa iz maks-hipa? Pošto se on nalazi u korenu, a drvo mora biti potpuno popunjeno na mesto izbačenog elementa je najjednostavnije upisati poslednji element iz hipa (najdešnji element poslednjeg nivoa). Ovim je zadovoljen uslov za raspored elemenata u drvetu, ali je svojstvo hipa moguće narušeno, jer taj element ne mora biti veći od svih ostalih. Popravku možemo izvršiti relativno jednostavno. Potrebno je da uporedimo vrednost u korenu sa vrednošću njegove dece. Ako je vrednost u korenu veća ili jednaka od tih vrednosti, onda koren zadovoljava uslov maks-hipa i procedura može da se završi. U suprotnom, menjamo vrednost u korenu sa većom od vrednosti njegove dece. Nakon toga koren zadovoljava uslov maks-hipa, i preostaje jedino da proverimo ono poddrvo u čijem je korenu završila vrednost korena. Ovo je problem istog oblika, samo manje dimenzije u odnosu na polazni i lako se rešava induktivno-rekurzivnom konstrukcijom. Broj koraka pomeranja naniže u najgorem slučaju odgovara visini drveta. Pošto u potpunom drvetu visine h može da stane 2^{h+1}-1 elemenata, visina logaritamski zavisi od broja elemenata. Vreme potrebno za uklanjanje najvećeg elementa iz hipa u kojem se nalazi n elemenata je *O(logn)*.

```
void pomeriNanize(vector<int>& hip, int k) {
    int najveci = k;
    int levo = levoDete(k), desno = desnoDete(k);
    if (levo < hip.size() && hip[levo] > hip[najveci])
        najveci = levo;
    if (desno < hip.size() && hip[desno] > hip[najveci])
        najveci = desno;
    if (najveci != k) {
        swap(hip[najveci], hip[k]);
        pomeriNanize(hip, najveci);
    }
}

int izbaciNajveci(vector<int>& hip) {
    hip[0] = hip.back();
    hip.pop_back();
    pomeriNanize(hip, 0);
}
```

Umetanje novog elementa funkcioniše po sličnom, ali obrnutom principu. Element je najjednostavnije ubaciti na kraj niza. Ipak, moguće je da mu tu nije mesto, jer je možda veći od svog roditelja. U tom slučaju moguće je izvršiti njihovu razmenu. Nakon zamene, poddrvo T sa korenom u novom čvoru zadovoljava uslov hipa i celo drvo bez poddrveta T takođe zadovoljava uslov hipa. Svakim pomeranjem novog elementa naviše, ostatak drveta bez poddrveta T se smanjuje i problem se svodi na problem manje dimenzije i rešava se induktivno-rekurzivnom konstrukcijom.

```
void pomeriNavise(vector<int>& hip, int k) {
    int r = roditelj(k);
    if (k > 0 && hip[k] > hip[r]) {
        swap(hip[k], hip[r]);
        pomeriNavise(hip, r);
    }
}

void ubaci(int x) {
    hip.push_back(x);
    pomeriNavise(hip, hip.size() - 1);
}
```

Još jedan način da implementiramo pomeranje naniže je da odredimo poziciju većeg od dva deteta i onda to dete uporedimo sa roditeljem.

```
void pomeriNanize(vector<int>& hip, int k) {
    int roditelj = k;
    int veceDete = levoDete(k);
    while (veceDete < hip.size()) {
```

```
        int desno = veceDete + 1;
        if (desno < hip.size() && hip[desno] > hip[veceDete])
            veceDete = desno;
        if (hip[roditelj] >= hip[veceDete])
            break;
        swap(hip[veceDete], hip[roditelj]);
        roditelj = veceDete;
        veceDete = levoDete(roditelj);
    }
}
```

5.11 Sortiranje uz pomoć hipa - implementacija, složenost

Algoritam sortiranja pomoću hipa (engl. HeapSort) predstavlja varijantu sortiranja selekcijom u kome se svi elementi niza postavljaju u maks-hip, a zatim se iz maks-hipa vadi jedan po jedan element i prebacuje se na kraj niza. Isti niz se može koristiti i za smeštanje polaznog niza i za smeštanje hipa i za smeštanje sortiranog niza. Time se štedi memorijski prostor, međutim, potrebno je malo prilagoditi funkcije za rad sa hipom. Funkcija za pomeranje naniže pored niza ili vektora u kome su smešteni elementi, mora kao parametar da primi i broj elemenata niza ili vektora koji predstavlja hip. Na osnovu tog broja je lako ustanoviti koji elementi niza pripadaju, a koji ne pripadaju hipu. Poređenje sa *hip.size()* se mora zameniti poređenjem sa tim brojem.
Jedan način da se od niza formira hip je da se krene od praznog hipa i da se jedan po jedan element ubacuje u hip.

```
void formirajHip(int a[], int n) {
    for (int i = 1; i < n; i++)
        pomeriNavise(a, i);
}
```

Invarijanta spoljne petlje je to da elementi na pozicijama [0, i) čine maks-hip. Pošto znamo da operacija pomeranja elementa naviše ispravno umeće poslednji element u postojeći maks-hip, lako je dokazati da invarijanta ostaje održana. Na kraju petlje je i = n, što znači da su svi elementi niza složeni u maks-hip. Dodatno, razmene ne menjaju multiskup elemenata niza. Ovaj način se naziva *formiranje hipa naniže* (Vilijamsov metod). Složenost najgoreg slučaja formiranja hipa jednaka je $\Theta(nlogn)$. Složenost operacije umetanja tj. pomeranja elementa naviše u hipu koji ima k elemenata jednaka je $O(logk)$, pa je ukupna složenost formiranja hipa asimptotski jednaka zbiru svih logaritama od 1 do n, što je $\Theta(nlogn)$.
Drugi, efikasniji način formiranja hipa naziva se *formiranje hipa naviše* (Flojdov metod). Ideja je da se elementi originalnog niza obilaze od pozadi i da se svaki element umetne u hip čiji koren predstavlja, tako što se spusti naniže kroz hip. Induktivna hipoteza u ovom pristupu je to da su svi elementi iz intervala (i, n) koreni ispravnih maks-hipova.

Kako proširiti invarijantu? Element na poziciji i ne mora da bude veći od svoje dece, ali znamo da su oba njegova deteta koreni ispravnih hipova. Stoga je samo potrebno spustiti element sa vrha na njegovo mesto, što je operacija koju smo već razmatrali prilikom operacije brisanja elementa iz hipa.

```
void formirajHip(int a[], int n) {
    for (int i = n/2; i >= 0; i--)
        pomeriNanize(a, n, i);
}
```

Veoma je zgodno to što je jedna polovina elemenata niza već na svom mestu. Spuštanje elemenata niz hip ima složenost *O(logn)*, pa je ovde ponovo u pitanju neki zbir logaritamskih složenosti i na prvi pogled može se pomisliti da će i ovde složenost biti $\Theta(nlogn)$ - ona će svakako biti *O(nlogn)*, ali ćemo pokazati da će biti niža tj. $\Theta(n)$. Nije svaki naredni logaritam u zbiru za jedan veći od prethodnog, kako je to bio slučaj prilikom pomeranja elemenata naviše. Ako bismo razmatrali hip u kome su svi nivoi potpuno popunjeni, postojala bi jedna pozicija sa koje bi se element spuštao celom visinom hipa, dve pozicije sa kojih bi se element spuštao visinom koja je za jedan manja, četiri elementa sa kojih je visina za 2 manja itd. Ukupan broj koraka za hip visine i bi bio: i + 2(i-1) + 4(i-2) + ... + 2^{i-1}·1.
Do ovog rezultata možemo doći i razmatranjem sledeće rekurentne jednačine. Broj koraka pomeranja naniže odgovara zbiru visina svih čvorova u drvetu. Neka je *H(i)* zbir svih visina potpunog binarnog drveta visine *i*. Tada je H(i) = 2H(i-1) + i, jer se potpuno binarno drvo sastoji od dva potpuna binarna drveta visine *i-1* i korena visine *i*. Važi H(0) = 0.
Od svih kompleksnih izvođenja bitan nam je samo krajnji rezultat, a to je da je H(i) = 2^{i+1}-i-2. Pošto je za potpuno drvo broj elemenata niza n = 2^i-1, važi da je H(i) = 2(n + 1)-i-2 = 2n-i i važi da je složenost konstrukcije naviše $\Theta(n)$. Dakle, formiranje hipa naviše ima asiptotski bolju složenost nego formiranje hipa naniže.
Implementacija faze sortiranja vađenjem elemenata hipa koja sledi nakon formiranja hipa: Invarijanta ove faze biće da se u nizu na pozicijama [0, i] nalaze elementi ispravno formiranog hipa, a da se u delu (i, n) nalaze sortirani elementi koji su svi veći ili jednaki od elemenata koji se trenutno nalaze u hipu. Na početku je i = n-1, pa je invarijanta trivijalno zadovoljena. U svakom koraku se najveći element hipa (element na poziciji 0) izbacuje iz hipa i dodaje na početak sortiranog dela niza (na poziciju i). Element sa pozicije i koji je ovom razmenom završio na vrhu hipa se pomera naniže, sve dok se zadovolji uslov hipa. Vrednost i se smanjuje za 1, čime se ispravno održava granica između hipa i sortiranog dela niza. Kada se petlja završi tada je $i = 0$, pa iz invarijante sledi da je niz ispravno sortiran (svi elementi na pozicijama (0, n) su sortirani i veći ili jednaki elementu na poziciji 0). Pošto se vrše samo razmene elemenata multiskup elemenata niza se ne menja.

```
void hipSort(int a[], int n) {
    formirajHip(a, n);
    for (int i = n-1; i > 0; i--) {
        swap(a[0], a[i]);
        pomeriNanize(a, i, 0);
    }
}
```

5.12 Heširanje - heš funkcije, osobine, složenost

Heš tabele se koriste za umetanje i traženje elemenata, a u nekim varijantama i za brisanje. Predstavljaju efikasnu strukturu podataka za implementaciju rečnika. Prosečno vreme izvršavanja operacije kojom se traži element u heš tabeli je *O(1)*.
Osnovna ideja: ako treba smestiti podatke sa ključevima iz opsega od 0 do *n-1*, a na raspolaganju je niz dužine n, onda se podatak sa ključem i smešta na poziciju i, $i = 0, 1, 2, ..., n-1$. Ovaj način smeštanja nazivamo *direktno adresiranje*. Možemo iskoristiti prednosti direktnog adresiranja sve dok možemo da priuštimo da alociramo niz koji ima jednu poziciju za svaku moguću vrednost ključa. U ovoj situaciji je trivijalno implementirati sve tri operacije rečnika.

```
DirektnoAdresiranje_Search(T, k)
ulaz: T - tabela, k - vrednost kljuca
izlaz: pokazivac na podatak iz tabele koji ima vrednost kljuca k
       ili NULL ako takav podatak ne postoji
1 return T[k]

DirektnoAdresiranje_Insert(T, x)
ulaz: T - tabela, x - podatak
izlaz: izmenjena tabela T
1 T[x.kluc] <- x

DirektnoAdresiranje_Delete(T, x)
ulaz: T - tabela, x - podatak
izlaz: izmenjena tabela T
1 T[x.kluc] <- NULL
```

Svaka od ovih operacija ima vreme izvršavanja *O(1)*.
Situacija se menja ako je opseg vrednosti ključeva od 1 do M toliko veliki da se niz dužine M ne može smestiti na računaru. Predpostavimo da je dato n ključeva iz skupa U veličine $|U| = M >> n$. Ideja je da ključeve smestimo u tabelu veličine m, tako da m nije mnogo veće od n. Potrebno je definisati funkciju: $h : \{0, 1, ..., M-1\} \longrightarrow \{0, 1, ..., m-1\}$ tzv. *heš funkciju* koja za svaki podatak određuje njegovu poziciju na osnovu vrednosti odgovarajućeg ključa.
Iako je veličina M skupa U mnogo veća od veličine tabele m, stvarni skup ključeva koje treba obraditi obično nije veliki.

Dobra heš funkcija preslikava ključeve ravnomerno po tabeli da bi se minimizirala mogućnost kolizija prouzrokovanih nagomilavanjem ključeva u nekim oblastima tabele. Heš funkcija treba da transformiše skup ključeva ravnomerno u skup slučajnih lokacija iz skupa $\{0, 1, ..., m-1\}$.
Uniformnost i *slučajnost* su bitne osobine heš funkcije.
Ako je veličina tabele m prost broj, a ključevi su celi brojevi, onda je jednostavna i dobra heš funkcija data izrazom: $h(x) = x \ mod \ m$.
Ako m nije prost broj, onda bi se ovakvom heš funkcijom izdvajalo najnižih k bitova ključa, što nije dobro. Ako su ovi ključevi parni brojevi, onda se pola heš tabele (sa neparnim indeksima) ne koristi. Moguće rešenje problema he da se koristi heš funkcija: $h(x) = (x \ mod \ p) \ mod \ m$, gde je p prost broj takav da je $m \ll p \ll M$.
Neugodna je situacija kad su svi ključevi oblika $r+kp$, za neki celi broj r, jer će svi ključevi imati istu vrednost heš funkcije r mod m. Tada ima smisla uvesti još jedan nivo randomizacije, time što bi se sama heš funkcija birala na slučajan način.

5.13 Heširanje - razrešavanje kolizija

Bez obzira na veličinu m tabele, dešavaće se da različitim ključevima odgovaraju iste lokacije. Ovakav nepoželjan događaj se zove *kolizija*. Potrebno je dakle rešiti dva problema: nalaženje heš funkicija koje minimiziraju verovatnoću pojave kolizija i postupak za obradu kolizija kad do njih ipak dođe.
Najjednostavniji način za obradu kolizije je tzv. *odvojeno nizanje*. U ovom pristupu svaki element heš tabele j pokazivač na povezanu listu koja sadrži sve ključeve smeštene na tu lokaciju u tabeli. Pri traženju zadatog ključa, najpre se izračunava vrednost heš funkcije, a zatim se adresirana povezana lista linearno pretražuje. Na prvi pogled upisivanje se pojednostavljuje umetanjem ključa na početak liste. Lista se ipak mora pregledati do kraja da bi se izbeglo upisivanje duplikata. Ovakav postupak je neefikasan ako su povezane liste dugačke. Poteškoće stvara i dinamička alokacija memorije, a potrebno je rezervisati i prostor za pokazivače. Element pronađen po vrednosti ključa se može obrisati za vreme *O(1)*. Pretraga zavisi od *faktora popunjenosti* heš tabele. Ako imamo heš tabelu T veličine m u kojoj je smečteno n ključeva, faktor popunjenosti je $\alpha = n/m$. U najgorem slučaju, svih n ključeva se preslikavaju u istu lokaciju tabele i kreiraju listu od n elemenata, pa je vreme izvršavanja pretrage *O(n)*.

```
OdvojenoNizanje_Insert(T, x)
ulaz: T - tabela, x - pokazivac na podatak
izlaz: izmenjena tabela T
1 if x se ne nalazi u listi T[h(x.kluc)]
2   umetni x na pocetak liste T[h(x.kluc)]

OdvojenoNizanje_Search(T, k)
ulaz: T - tabela, k - vrednost kljuca koja se trazi
izlaz: podatak iz tabele koji ima vrednost kljuca k
       ili NULL ako takav podatak ne postoji
```

```
1 trazi element sa kljucem k u listi T[h(k)]

OdvojenoNizanje_Delete(T, x)
ulaz: T - tabela, x - pokazivac na podatak koji treba obrisati
izlaz: izmenjena tabela T
1 obrisi x iz liste T[h(x.kluc)]
```

Drugi način za obradu kolizije predstavlja *otvoreno adresiranje*. Svi elementi se smeštaju u niz - heš tabelu. Svaki element heš tabele sadrži ili ključ ili NULL. Prilikom potrage za ključem, sistemski se pretražuju pozicije tabele dok se ne nađe traženi ključ ili se ustanovi da on nije tu. Da bi se element smestio u tabelu, sukcesivno se ispituju pozicije heš tabele dok se ne pronađe prazna pozicija na koju se smešta ključ.
Najjednostavniji metod otvorenog adresiranja je *linearno pupunjavanje*. Ako je lokacija sa adresom *h(x)* već zauzeta, onda se novi element sa ključem x zapisuje na susednu lokaciju $(h(x)+1) \bmod m$ (lokacija koja sledi iza *m-1* je 0). Prilikom traženja zadatog ključa x neophodno je linearno pretraživanje počevši od lokacije *h(x)* do nepopunjene lokacije. Ovo rešenje je efikasno ako je tabela relativno retko popunjena. Inače će doći do mnogo *sekundarnih kolizija* - kolizija prouzrokovanih ključevima sa različitim vrednostima heš funkcije. Pri linearnom popunjavanju se brisanja ne mogu izvesti korektno. Ako su potrebna i brisanja, mora se koristiti neki od metoda za obradu kolizija koji koristi pokazivače ili se mora koristiti mogućnost "lenjog brisanja".

```
LinearnoPopunjavanje_Insert(T, k)
ulaz: T - tabela, k - vrednost kljuca
izlaz: indeks pozicije na kojoj je zapisan kljuc k i imenjena tabela T
1 i <- 0
2 repeat
3    j <- (h(k)+i) mod m
4    if T[j] = NULL
5        T[j] <- k
6        return j
7    else i <- i+1
8 until i = m
9 error "prekoracenje hes tabele"

LinearnoPopunjavanje_Search(T, x)
ulaz: T - tabela, x - podatak
izlaz: indeks kljuca k u tabeli T ili NULL inace
1 i <- 0
2 repeat
3    j <- (h(k)+i) mod m
4    if T[j] = k
5        return j
6    i <- i+1
7 until T[j] != NULL or i = m
8 return NULL
```

Svaki novi ključ preslikan u *i*, *i+1* ili *(i+2)* mod *m*, ne samo da izaziva novu koliziju, nego i povećava veličinu ovog popunjenog odsečka što kasnije izaziva još više sekundarnih kolizija - *efekat grupisanja*. Efekat grupisanja se može ublažiti *dvostrukim heširanjem*. Kad dođe do kolizije pri upisu elementa x na poziciju i, h(x) = i, izračunava se vrednost druge heš funkcije h_2(x), pa se x smešta na prvu slobodnu među lokacijama $(i+h_2(x)) \bmod m$, $(i+2h_2(x)) \bmod m$,... umesto $(i+1) \bmod m$, $(i+2) \bmod m$... Korak sa kojim tražimo novu poziciju nije fiksan već zavisi od samog ključa.

```
DvostrukoHesiranje_Insert(T, k)
ulaz: T - tabela, k - vrednost kljuca
izlaz: indeks pozicije na kojoj je zapisan kljuc k i izmenjena tabela
1 i <- 0
2 repeat
3    j <- (h1(k) + i*h2(k)) mod m
4    if T[j] = NULL
5        T[j] <- k
6        return j
7    else i <- i+1
8 until i = m
9 error "prekoracenje hes tabele"

DvostrukoHesiranje_Search(T, x)
ulaz: T - tabela, x - podatak
izlaz: indeks pozicije u tabeli na kojoj je pronadjen kljuc k ili NULL
1 i <- 0
2 repeat
3    j <- (h1(k) + i*h2(k)) mod m
4    if T[j] = k
5        return j
6    i <- i+1
7 until T[j] != NULL or i = m
8 return NULL
```

6 Primene sortiranja i binarne pretrage

6.1 Sortiranje - bibliotečka podrška, algoritmi, primene

Funkcijom ***sort*** vrši se sortiranje neke kolekcije (niza, deka, liste). Parametri su par iteratora koji ograničavaju poluotvoreni raspon niza koji se sortira - prvi iterator ukazuje na prvi element, a drugi iterator neposredno iza poslednjeg elementa. Moguće je navesti i treći parametar koji predstavlja funkciju poređenja.

Pravedna podela čokolada

Problem: Dato je n paketa čokolade i za svaki od njih je poznato koliko čokoladica sadrži. Svaki od k učenika uzima tačno jedan paket, pri čemu je cilj da svi učenici imaju što približniji broj čokoladica. Kolika je najmanja moguća razlika između onog učenika koji uzme paket sa najmanje i onog koji uzme paket sa najviše čokoladica?
Najdirektniji način da se reši zadatak je da se ispitaju svi podskupovi od k elemenata skupa od n elemenata i da se među njima odabere najbolji, $O(n^k)$.
Kada se polazni paketi sortiraju po broju čokoladica, učenici treba da uzmu uzastopnih k paketa.

```
int pravednaPodela(const vector<int>& a) {
    sort(begin(a), end(a));
    int min = numeric_limits<int>::max();
    for (int i = 0; i + k - 1 < n; i++) {
        int razlika = a[i + k - 1] - a[i];
        if (razlika < min)
            min = razlika;
    }
}
```

Najduža doktorova pauza

Problem: Poznata su vremena dolaska pacijenata na pregled i vreme trajanja njihovog pregleda. Kolika je najveća pauza koju doktor može imati u toku tog dana, između pregleda dva svoja pacijenta?
Nakon sortiranja tražimo najveću razliku između vremena odlaska nekog pacijenta i vremena dolaska sledećeg.

```
int najvecaPauza(vector<int>& dosao, vector<int>& trajao) {
    sort(begin(dosao), end(dosao));
    int maxPauza = 0;
    for (int i = 0; i < n-1; i++) {
        int pauza = dosao[i+1] - (dosao[i] + trajao[i]);
        if (pauza > maxPauza)
            maxPauza = pauza;
    }
    return maxPauza;
}
```

Najbliže sobe

Problem: Dva gosta su došla u hotel i žele da odsednu u sobama koje su što bliže jedna drugoj, da bi tokom večeri mogli da zajedno rade u jednoj od tih soba. Ako postoji više takvih soba, oni biraju da budu što dalje od recepcije, tj. u sobama sa što većim rednim brojevima, kako im buka ne bi smetala. Ako je poznat spisak slobodnih soba u tom trenutku, napiši program koji određuje brojeve soba koje gosti treba da dobiju.
Direktan pristup rešenju bi bio da se izračunaju rastojanja između svake dve sobe i da se pronađe par sa najmanjim rastojanjima, $O(n^2)$.
Bolje rešenje se može dobiti ako se niz pre toga sortira. Najbliži element svakom elementu u sortiranom nizu je jedan od njemu susednih. Dakle, ako broj a učestvuje u paru najbližih soba, onda drugi element tog para može biti ili onaj broj koji je neposredno ispred a u sortiranom redosledu ili onaj koji je neposredno iza njega. Zato je nakon sortiranja dovoljno proveriti sve razlike između susednih elemenata i odrediti najmanju od njih (ako ima više istih, određujemo poslednju), $O(nlogn)$.

```
pair<int, int> najblizeSobe(vector<int>& sobe) {
    sort(begin(sobe), end(sobe));
    int min = 0;
    for (int i = 1; i < n-1; i++)
        if (sobe[i+1] - sobe[i] <= sobe[min+1] - sobe[min])
            min = i;
    return make_pair(sobe[min], sobe[min+1]);
}
```

Najduži podskup uzastopnih brojeva

Problem: U nizu celih brojeva odrediti najbrojniji podskup elemenata koji se mogu urediti u niz uzastopnih celih brojeva. Ako ima više takvih podskupova, prikazati prvi (onaj u kojem su brojevi najmanji).
Kada je niz sortiran i nema duplikata, pronalaženje traženog podskupa se svodi na pronalaženje najdužeg segmenta niza koji čine uzastopni brojevi.

```
int najduziPodskupUzastopnih(int a[], int n) {
    sort(a, next(a, n));
    n = distance(a, unique(a, next(a, n)));
    int duzinaTekuceSerije = 1, duzinaMaxSerije = 1;
    for (int i = 1; i < n; i++) {
        if (a[i] == a[i - 1] + 1)
            duzinaTekuceSerije++;
        else
            duzinaTekuceSerije = 0;
        if (duzinaTekuceSerije > duzinaMaxSerije)
            duzinaMaxSerije = duzinaTekuceSerije;
    }
    return duzinaMaxSerije;
}
```

Najbrojniji presek intervala

Problem: Ljudi su dolazili i odlazili sa bazena i za svakog posetioca je poznato vreme dolaska i vreme odlaska (pretpostavićemo da je čovek na bazenu u periodu oblika [a, b), tj. da ako jedan čovek ode, a drugi dođe u apsolutno istom trenutku, da se broj ljudi ne menja). Koliko je ljudi najviše bilo istovremeno na bazenu?
Broj ljudi koji su trenutno na bazenu se menja samo u trenucima kada neko dođe ili kada neko ode. Da bi se odredio najveći broj ljudi dovoljno je razmotriti samo te karakteristične trenutke. Veoma prirodno je da te karakteristične trenutke obrađujemo hronološki, u rastućem redosledu vremena. Možemo kreirati niz koji sadrži sve karakteristične trenutke i za svaki trenutak beležiti da li je dolazni ili odlazni trenutak. Taj niz možemo sortirati i zatim obrađivati redom, izračunavajući za svaki trenutak broj ljudi na bazenu inkrementalno, na osnovu broja ljudi na bazenu u prethodnom karakterističnom trenutku. Ako u nekom vremenskom trenutku više ljudi dolazi ili odlazi, broj ljudi ćemo upoređivati sa maksimumom tek kada obradimo sve ljude koji su došli ili otišli u tom trenutku.

```
int najbrojnijiPresek(const vector<pair<int, int>>& intervali){
    int n = intervali.size();
    vector<pair<int, int>> promene(2*n);
    for (int i = 0; i < n; i++) {
        promene[2*i] = make_pair(intervali[i].first, 1);
        promene[2*i+1] = make_pair(interval[i].second, -1);
    }
    sort(begin(promene), end(promene));
    int trenutnoPrisutno = 0;
    int maksPrisutno = 0;
    int i = 0;
    while (i < n) {
        trenutnoPrisutno += promene[i++].second;
        if (trenutnoPrisutno > maksPrisutno)
            maksPrisutno = trenutnoPrisutno;
    }
    return maksPrisutno;
}
```

Dužina pokrivača

Problem: Dato je n zatvorenih intervala realne prave. Odredi ukupnu dužinu koju pokrivaju, kao i najmanji broj zatvorenih intervala koji pokrivaju isti skup tačaka kao i polazni intervali (oni se mogu dobiti ukrupnjavanjem polaznih intervala).
Prvo se sortira niz svih granica intervala, kojima je pridružena oznaka da li predstavljaju početak ili kraj. Potom se formira niz ukrupnjenih intervala, koji čine pokrivač skupa tačaka pokrivenih polaznim intervalima.

```
typedef vector<pair<double, double>> Intervali;
 void ukrupniIntervale(const Intervali& ulazni, Intervali& izlazni) {
    vector<pair<double, int>> promene(ulazni.size()*2);
    for(int i = 0; i < ulazni.size(); i++) {
        promene[2*i] = make_pair(ulazni[i].first, 1);
        promene[2*i+1] = make_pair(ulazni[i].second, -1);
    }
    sort(begin(promene), end(promene));
    bool zapocetIzlazni = false;
    double xPocetakIzlaznog = 0;
    int brojUlaznih = 0;
    int i = 0;
    while(i < promene.size()) {
        double xTrenutno = promene[i].first;
        while (i < promene.size() && promene[i].first == xTrenutno)
            brojUlaznih += promene[i++].second;
        if (!zapocetIzlazni && brojUlaznih > 0) {
            zapocetIzlazni = true;
            xPocetakIzlaznog = xTrenutno;
        }
        if (zapocetIzlazni && brojUlaznih == 0) {
            izlazni.push_back(make_pair(xPocetakIzlaznog, xTrenutno));
            zapocetIzlazni = false;
        }
    }
 }
```

Zbir minimuma trojki

Problem: Dat je niz pozitivnih celih brojeva a_0, a_1,..., a_{n-1}. Za svaku trojku $0 \leq i < j < k \leq n$ odrediti najmanju vrednost od tri broja a_i, a_j, a_k, a zatim odrediti zbir tako dobijenih vrednosti.
Ključna opaska u zadatku je da zbir minimuma svih trojki elemenata niza ne zavisi od redosleda elemenata u nizu. Indeksi $0 \leq i < j < k < n$, i elementi na odgovarajućim pozicijama određuju zapravo sve tročlane (multi)podskupove skupa elemenata niza. Permutovanjem niza, njegovih tročlani (multi)podskupovi se ne menjaju. Određivanje traženog zbira najjednostavnije je ako je niz sortiran.

```
int brojParova(int k) { return k * (k - 1) / 2; }

 int zbirMinimumaTrojki(vector<int>& a) { // O(n)
    sort(begin(a), end(a));
    int zbir = 0;
    for (int i = 0; i < n - 2; i++) {
        int brojTrojki = brojParova(n - 1 - i);
        zbir += brojTrojki * a[i]
    }
    return zbir; }
```

Odlični takmičari

Problem: Studenti su se na jednom turniru takmičili iz programiranja i matematike. Takmičar je odličan ako ne postoji takmičar koji je od njega osvojio strogo više poena i iz programiranja i iz matematike. Napisati funkciju koja određuje ukupan broj odličnih takmičara.
Rešenje grubom silom daje složenost najgoreg slučaja $O(n^2)$.
Sortiranjem lako možemo za svakog takmičara da odredimo one koji od njega imaju više poena iz matematike kao i one koji imaju manje poena od njega iz matematike. On će dakle, biti odličan akko niko desno od njega nema strogo više poena iz programiranja. Znamo da maksimum možemo računati inkrementalno. Dakle, takmičare možemo sortirati u nerastućem broju poena iz matematike, a one koji imaju isti broj poena iz matematike u neopadajućem broju poena iz programiranja. Redom obilazimo sortirani niz i održavamo maksimum poena iz programiranja do sada obrađenih takmičara. Ako tekući takmičar ima više ili jednako poena od tekućeg maksimuma, on je odličan, pa uvećavamo broj odličnih takmičara i ažuriramo maksimum.

```
struct Poeni {
    int mat;
    int prog;
};

void odlicniTakmicari(const vector<Poeni>& poeni) {
    sort(begin(poeni), end(poeni), [](auto& p1, auto& p2) {
        return p1.mat > p2.mat || p1.mat == p2.mat
               && p1.prog < p2.prog;
    });
    int brojOdlicnih = 0;
    int maxProg = 0;
    for (int i = 0; i < n; i++) {
        if (poeni[i].prog >= maxProg) {
            brojOdlicnih++;
            maxProg = prog;
        }
    }
    return brojOdlicnih;
}
```

H-indeks

Problem: Rangiranje naučnika vrši se pomoću statistike koja se naziva Hiršov indeks ili kraće h-indeks. H-indeks je najveći broj h takav da naučnik ima bar h radova sa bar h citata. Napisati program koji na osnovu broja citata svih radova naučnika određuje njegov h-indeks.
Jedan način da se zadatak efikasno reši je da se radovi sortiraju u nerastućem broju citata i onda da se redom proverava da li h-ti po redu rad ima bar h citata.

```
int hIndeks(const vector<int>& brojCitata) {
    sort(brojCitata.begin(), brojCitata.end(), greater<int>());
    int indeks = 0;
    while (indeks < n && brojCitata[indeks] > indeks)
        indeks++;
    return indeks;
}
```

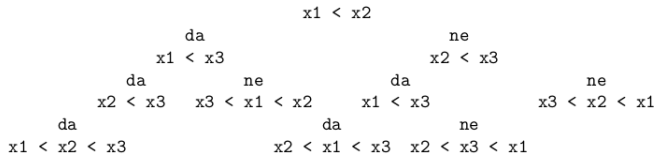
Druga mogućnost je da se za svaki broj h izračuna tačan broj radova B_h koji imaju bar h citata. Naivan način da se to uradi je da se da se za svaki broj h prolazi kroz niz (koji ne mora biti sortiran) i da se odredi broj elemenata niza koji su veći ili jednaki h. Poboljšanje dolazi uz pomoć inkrementalnosti. Možemo primetiti da se broj radova koji imaju bar h citata može izračunati kao zbir broja radova koji imaju više od h citata tj. broja radova koji imaju bar $h+1$ citat i broja radova koji imaju tačno h citata, tj. $B_h = B_{h+1} + b_h$, gde je b_h broj radova koji imaju tačno h citata.

```
int hIndeks(const vector<int>& brojCitata) {
    vector<int> brojRadova(n + 1, 0);
    for (int i = 0; i < n; i++) {
        int brojCitata;
        cin >> brojCitata;
        broj_radova[broj_citata < n ? broj_citata : n]++;
    }
    int indeks = n;
    int ukupnoRadova = brojRadova[n];
    while (ukupnoRadova < indeks) {
        indeks--;
        ukupnoRadova += brojRadova[indeks];
    }
    cout << h_indeks << endl;
}
```

Algoritam sortiranja se naziva *stabilnim* (engl. stable) ako se prilikom sortiranja zadržava originalni redosled elemenata sa istim ključem. Uobičajene implementacije *sortiranja selekcijom*, *brzog sortiranja* i *sortiranja pomoću hipa* nisu stabilne. Sa druge strane, uobičajene implementacije *sortiranja umetanjem* i *sortiranja objedinjavanjem* jesu stabilne. Kod sortiranja umetanjem bitno je zaustaviti se čim tekući element stigne iza elementa koji ima isti ključ kao i on. Kod sortiranja objedinjavanjem, za stabilnost je bitno da se tokom objedinjavanja u slučaju istih ključeva u levoj i desno polovini bira element iz leve polovine. Svaki se algoritam može napraviti stabilnim po cenu korišćenja dodatne memorije i malo dodatnog vremena. Funkcija *sort* u jeziku C++ ne garantuje stabilnost. Ako nam je stabilnost potrebna, na raspolaganju imamo funkciju *stable_sort*, koja je obično zasnovana na algoritmu *MergeSort*. Ako na raspolaganju imamo dodatno *O(n)* memorije tada je njeno vreme izvršavanja u najgorem slučaju *O(nlogn)*, a u suprotnom je $O(n(\log n)^2)$.

6.2 Sortiranje - donja granica složenosti

Algoritmi sortiranja koje smo do sada sreli su imali vremensku složenost $O(n^2)$ ili $O(nlogn)$. Postavlja se pitanje da li je moguće napraviti algoritam koji bi imao manju složenost. Ako se fokusiramo na algoritme koje sortiranje vrše samo na osnovu upoređivanja međusobnih elemenata niza (a takvi su svi algoritmi koje smo do sada susreli), možemo formalno da dokažemo da je donja granica složenosti takvih algoritama $\Theta(nlogn)$.
Posmatraćemo model *drveta odlučivanja* (engl. decision tree). To je teorijski model izračunavanja koji se predstavlja drvetom u kojem unutrašnji čvorovi predstavljaju pitanja koja se postavljaju i u kom se na osnovu odgovora na pitanje izvršavanje programa usmerava levo ili desno. U slučaju algoritama sortiranja pitanja će podrazumevati poređenje proizvoljna dva elementa niza. Listovi drveta predstavljaju tražene rezultate. U slučaju algoritama sortiranja, svaki list drveta predstavlja jedan mogući raspored (redosled) elemenata koji se sortiraju. Određivanje rasporeda n elemenata drvetom odlučivanja, tj. samo upoređivanjem parova elemenata niza zahteva bar $\Theta(nlogn)$ upoređivanja.



6.3 Sortiranje prebrojavanjem

Kada se zna da svi elementi niza dolaze iz nekog relativno uskog intervala vrednosti, sortiranje je moguće uraditi i efikasnije nego kada se koristi sortiranje upoređivanjem. Jedna ideja je da se sortiranje vrši *prebrojavanjem* tj. određivanjem broja pojavljivanja svake vrednosti iz dopuštenog intervala vrednosti i da se onda niz rekonstruiše tj. popuni iz početka na osnovu izračunatog broja pojavljivanja. Ako je unapred poznato da je interval vrednosti interval [0, m], tada za prebrojavanje možemo upotrebiti običan niz brojača. Složenost faze prebrojavanja je $O(n)$, a složenost rekonstrukcije je $O(m)$, pa je ukupna složenost jednaka $O(n + m)$.

```
void sortiranjePrebrojavanjem(vector<int>& a) {
    int m = *max_element(begin(a), end(a));
    vector<int> frekvencije(m+1, 0);
    for (int i = 0; i < n; i++)
        frekvencije[a[i]]++;
    int k = 0;
    for (int j = 0; j < m; j++)
        for (int i = 0; i < frekvencije[j]; i++)
            a[k++] = j;
}
```

6.4 Sortiranje razvrstavanjem

Ideja prebrojavanja se može i malo uopštiti. Najefikasnija varijanta sortiranja nastupa kada svakom elementu unapred znamo mesto u nizu na kom treba da se nađe. Tada je dovoljno proći kroz originalni niz i svaki element samo upisati na njegovo mestu u rezultujućem nizu. Nekada više elemenata treba smestiti na isto mesto. Na primer, ako želimo da sortiramo pisma prema odredištu, tada je dovoljno napraviti onoliko pregrada koliko postoji mogućih odredišta i svako pismo smestiti u odgovarajuću pregradu. Pregrade bi se mogle implementirati kao neka dvodimenzionalna struktura. Ako želimo da upotrebimo običan niz, tada bi sva pisma koja idu na istu destinaciju trebalo smestiti jedno iza drugog. Da bismo znali poziciju na kojoj počinju pisma za svaku destinaciju, dovoljno je da prebrojimo koliko ukupno pisama treba dostaviti na sve destinacije pre tekuće. Centralni korak u algoritmu sortiranja je prebrojavanje broja elemenata niza koji odgovaraju svakom mogućem ključu. Ovakav postupak sortiranja je *sortiranje razvrstavanjem*.

Problem: Državna komisija je napravila spisak svih birača u državi. Potrebno je da se svakoj opštini distribuira spisak birača sa teritorije te opštine, ali tako da redosled ostane isti kakav je na polaznom spisku državne komisije. Na standardni izlaz ispisati spiskove za sve opštine, svaki u posebnom redu. Opštine treba da budu uređene leksikografski, rastuće.

```
struct Birac {
    string opstina;
    string sifra;
};

void sortiraj(vector<Birac>& biraci) {
    map<string, int> brojBiraca;
    for (auto birac : biraci)
        brojBiraca[birac.opstina]++;
    map<string, int> pozicije;
    int prethodnoBiraca = 0;
    for (auto it : brojTakmicara) {
        pozicije[it.first] = prethodnoBiraca;
        prethodnoBiraca += it.second;
    }
    vector<Birac> sortirano(biraci.size());
    for (auto birac : biraci) {
        sortirano[pozicije[birac.opstina]] = birac;
        pozicije[birac.opstina]++;
    }
    biraci = sortirano;
}
```

6.5 Sortiranje višestrukim razvrstavanjem (Radix sort)

Kada je broj mogućih vrednosti ključa k veliki, tada je i prostorna i vremenska složenost nepovoljna. U nekim situacijama situaciju je moguće popraviti tako što se razvrstavanje vrši u više faza.

Problem: Potrebno je da niz učenika sortiramo na osnovu inicijala, ali tako da se unutar svake grupe učenika sa istim inicijalima raspored ostane isti kao na originalnom spisku.
Jedan način da se zadatak reši je da se uradi sortiranje prebrojavanjem, na osnovu svih mogućih inicijala. Za prebrojavanje učenika na osnovu svakog mogućeg inicijala možemo koristiti ili mapu ili niz, ali je tada potrebno obezbediti preslikavanje inicijala u redni broj elemenata niza. Umesto toga, zadatak je moguće rešiti primenom sortiranja prebrojavanjem u dve faze. Učenici se mogu prvo sortiranjem prebrojavanjem sortirati u 26 grupa na osnovu prezimena, a zatim u 26 grupa na osnovu imena. Ključni argument u dokazu korektnosti ovog dvofaznog postupka je to da je sortiranje prebrojavanjem stabilno, odnosno to da se zadržava originalni redosled elemenata sa istim ključem. Sva imena koja počinju na isto početno slovo zadržavaju redosled pre sortiranja po imenu, a pošto su prethodno bila sortirana po prezimenu, nakon sortiranja po imenu, biće sortirani na osnovu inicijala. Pošto tokom sortiranja po prezimenu učenici sa istim početnim slovom prezimena zadržavaju polazni redosled, nakon sortiranja po imenu, učenici sa istim inicijalima biće poređani u originalnom redosledu.

```
enum {IME = 0, PREZIME = 1};
typedef array<string, 2> Ucenik;

vector<Ucenik> razvrstaj(const vector<Ucenik>& ucenici, int ip) {
    int brojPojavljivanja[26] = {0};
    for (auto ucenik : ucenici)
        brojPojavljivanja[ucenik[ip][0] - 'a']++;
    int pozicija[26];
    pozicija[0] = 0;
    for (int i = 1; i < 26; i++)
        pozicija[i] = pozicija[i-1] + brojPojavljivanja[i-1];
    vector<Ucenik> rezultat(ucenici.size());
    for (auto ucenik : ucenici)
        rezultat[pozicija[ucenik[ip][0] - 'a']++] = ucenik;
    return rezultat;
}
```

Problem: Dati niz prirodnih brojeva sortirati primenom sortiranja višestrukim razvrstavanjem.

```
void sortiranjeRazvrstavanjem(vector<int>& a, int s) {
    int broj[10] = {0};
    for (int i = 0; i < a.size(); i++)
        broj[(a[i] / s) % 10]++;
```

```
    for (int i = 1; i < 10; i++)
        broj[i] += broj[i-1];
    vector<int> pom(a.size());
    for (int i = a.size() - 1; i >= 0; i--)
        pom[--broj[(a[i] / s) % 10]] = a[i];
    a = pom;
}

void sortiranjeVisestrukimRazvrstavanjem(vector<int>& a) {
    int max = numeric_limits<int>::min();
    for (int x : a)
        if (x > max)
            max = x;
    for (int s = 1; max / s > 0; s *= 10)
        sortiranjeRazvrstavanjem(a, s);
}
```

6.6 Binarna pretraga - bibiliotečka podrška, implementacija, primene

Funkcija *binary_search* vrši binarnu pretragu niza i vraća logičku vrednost tačno akko je traženi element prisutan u rasponu koji se pretražuje. Naravno, kao i kod svih varijanti binarne pretrage, potrebno je da je raspon koji se pretražuje sortiran. Parametri su par iteratora koji ograničavaju poluotvoreni raspon i element koji se pretražuje. Funkciju poređenja moguće je navesti kao četvrti argument funkcije.
Funkcija *equal_range* vrši binarnu pretragu i vraća par iteratora koji ograničavaju poluotvoreni raspon u kom se nalaze svi elementi ekvivalentni datom. Parametri su isti kao i kod prethodne funkcije.
Funkcija *lower_bound* vraća iterator na prvi element koji je veći ili jednak datom.
Funkcija *upper_bound* vraća iterator na prvi element koji je strogo veći od datog.

Element na svojoj poziciji

Problem: Napisati funkciju koja proverava da li u strogo rastućem nizu elemenata postoji pozicija i takva da se na poziciji i nalazi vrednost i tj. da važi da je $a_i = i$.

```
int itiNaMestuI(vector<int>& a) {
    for (int i = 0; i < n; i++)
        a[i] -= i;
    auto it = lower_bound(a.begin(), a.end(), 0);
    if (it != a.end() && *it == 0)
        return distance(a.begin(), it);
    else
        return -1; }
```

Broj kvadrata

Problem: Dat je skup od n tačaka u ravni ca celobrojnim koordinatama. Napisati program koji određuje koliko se različitih kvadrata može napraviti tako da su im sva četiri temena u tom skupu tačaka.
Rešenje grubom silom je da se proveri svaka četvorka tačaka i utvrdi da li čini, $O(n^4)$. Bolji način je da se za svaki par tačaka proveri da li čini dijagonalu kvadrata sastavljenog od tačaka iz tog skupa. Ako dobijena temena kvadrata nisu celobrojna, možemo ih odmah eliminisati. U suprotnom, potrebno je da proverimo da li se nalaze u polaznom skupu tačaka. Naivni način je da se svaki put izvrši linearna pretraga. Mnogo bolje je da se tačke sortiraju nekako i da se primeni binarna pretraga. Pošto će svaki kvadrat biti pronađen dva puta dobijeni broj kvadrata treba podeliti sa dva. Umesto razmatranja dijagonala, moguće je proveriti i da li svaki par tačaka čini stranicu kvadrata, ali tada bi se svaki kvadrat pronašao 4 puta.

```
typedef pair<int, int> Tacka;

Tacka transliraj(const Tacka& t, int dx, int dy) {
    return make_pair(t.first + dx, t.second + dy);
}

bool DrugaDvaTemenaKvadrata(const Tacka& t1, const Tacka& t2,
                            Tacka& t3, Tacka& t4) {
    int x1 = t1.first, y1 = t1.second, x2 = t2.first, y2 = t2.second;
    int dx = x2 - x1, dy = y2 - y1;
    if ((dx + dy) % 2 != 0)
        return false;
    t3 = transliraj(t1, (dx - dy) / 2, (dy + dx) / 2);
    t4 = transliraj(t1, (dx + dy) / 2, (dy - dx) / 2);
    return true;
}

int brojKvadrata(const vector<int>& tacke) {
    sort(tacke.begin(), tacke.end());
    int broj = 0;
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++) {
            Tacka t3, t4;
            if (DrugaDvaTemenaKvadrata(tacke[i], tacke[j], t3, t4))
                if (binary_search(tacke.begin(), tacke.end(), t3) &&
                    binary_search(tacke.begin(), tacke.end(), t4))
                        broj++;
        }
    return broj / 2;
}
```

Broj studenata iznad praga

Problem: Potrebno je odrediti prag poena za upis na fakultet. Komisija je stalno suočena sa pitanjima koliko bi se studenata upisalo, kada bi prag bio jednak datom broju. Napisati program koji učitava poene svih kandidata i efikasno odgovara na upite ovog tipa.
Za svaki uneti prag potrebno je moći efikasno ispitati koliko ima takmičara čiji su poeni iznad praga. Ako se niz poena sortira nakon učitavanja, brojanje takmičara iznad praga se može ostvariti efikasno. Kada se pronađe pozicija prvog takmičara čiji su poeni iznad praga, znamo da su svi takmičari od te pozicije pa do kraja niza primljeni. Prvu vrednost veću ili jednaku datoj u sortiranom nizu možemo jednostavno odrediti binarnom pretragom, pa je složenost ovog pristupa jednaka $O((n + m)logn)$.

```
int n;
cin >> n;
vector<int> poeni(n);
for (int i = 0; i < n; i++)
    cin >> poeni[i];
sort(begin(poeni), end(poeni));
int m;
cin >> m;
for (int i = 0; i < m; i++) {
    int prag;
    cin >> prag;
    auto it = lower_bound(begin(poeni), end(poeni), prag);
    cout << distance(it, end(poeni)) << endl;
}
```

Kružne zone

Problem: Prostor je podeljen u zone oblika kružnih prstenova, pri čemu širine prstenova mogu biti međusobno različite. Napiši program koji učitava broj i širinu svih zona i za niz učitanih tačaka određuje zonu kojoj pripadaju.
Tačka pripada nekoj zoni akko je njeno rastojanje od koordinatnog početka strogo veće od unutrašnjeg, a manje ili jednako spoljašnjem prečniku te zone. Unutrašnji prečnik neke zone jednak je zbiru širina svih zona pre nje, a spoljašnji prečnik je jednak zbiru širina svih zona pre nje, uključujući i njenu širinu. Unutrašnji prečnik neke zone jednak je spoljašnjem prečniku prethodne zone. Dovoljno je da izračunamo spoljašnje prečnike svih zona, a oni se mogu izračunati kao parcijalni zbirovi širina zona. Njih možemo računati inkrementalno ili bibliotečkom funkcijom *partial_sum*. Kada je poznat niz spoljašnjih prečnika zona, zona u kojem se tačka nalazi se može naći tako što se nađe prvi spoljašnji prečnik zone koji je veći ili jednak rastojanju tačke od koordinatnog početka. To možemo uraditi binarnom pretragom.

```
void indeksiZona(const vector<double>& zone,
  const vector<pair<double, double>>& tacke, vector<int>& indeksi) {
    vector<double> poluprecnici(zone.size());
    partial_sum(zone.begin(), zone.end(), poluprecnici.begin());
    for (int i = 0; i < tacke.size(); i++) {
        int x = tacke[i].first, y = tacke[i].second;
        double r = sqrt(x*x + y*y);
        auto it = lower_bound(poluprecnici.begin(),
                              poluprecnici.end(), r);
        if (it == poluprecnici.end())
            indeksi[i] = -1;
        else
            indeksi[i] = distance(poluprecnici.begin(), it);
    }
}
```

Broj najbliži datom

Problem: Dat je sortirani niz od n različitih brojeva. Odrediti broj najbliži datom (ako su dva podjednako udaljena, vratiti manji od njih).

```
int najbliziDatom(const vector<int>& a, int x) { // O(log n)
    auto it = lower_bound(begin(a), end(a), x);
    if (it == end(a))
        return *prev(it);
    else if (it == begin(a))
        return *it;
    else{
        if (*it - x < x - *prev(it))
            return *it;
        else
            return *prev(it);
    }
}
```

6.7 Optimizacija korišćenjem binarne pretrage - primeri

Štucajući podniz

Problem: Ako je s niska, onda je s^n niska koja se dobija ako se svako slovo ponovi n puta. Napisati program koji određuje najveći broj n takav da je s^n podniz niske t.

```
bool jeStucajuci(const string& podniz, const string& niz, int n) {
    int i = 0;
    for (char c : podniz) {
        for (int k = 0; k < n; k++) {
            while (i < niz.size() && niz[i] != c)
                i++;
```

```
            if (i == niz.size())
                return false;
            i++;
        }
    }
    return true;
}

int najduziStucajuciPodniz(const string& podniz, const string& niz) {
    int l = 0, d = niz.size() / podniz.size();
    while (l <= d) {
        int s = l + (d - l) / 2;
        if (jeStucajuci(podniz, niz, s))
            l = s + 1;
        else
            d = s - 1;
    }
    return d;
}
```

Drva

Problem: Drvoseča treba da naseče n metara drva. On ima testeru koja može da se podesi na određenu visinu i da poseče sve što je iznad te visine. Odredi najvišu moguću visinu na koju može da podesi testeru tako da naseče dovoljno drva, ali ne više od toga. Poznat je broj i visina svakog drveta u metrima. Testera se može podešavati takođe sa preciznošću jednog metra.
Jedno rešenje problema se može zasnovati na binarnoj pretrazi po rešenju. Za fiksiranu visinu testere u vremenu $O(n)$ možemo izračunati ukupnu količinu nasečenog drveta. Binarna pretraga je primenljiva jer znamo da je do određenih visina testere drveta dovoljno, a da je od određene visine testere drveta premalo, tako da zapravo tražimo prelomnu tačku, tj. najveću visinu testere za koju je drveta dovoljno tj. poslednji element niza koji zadovoljava uslov. Ako je maksimalna visina drveta M, tada je složenost ovog pristupa $O(nlogM)$.

```
int testera(vector<int>& visine, int potrebno) {
    int od_visina = 0;
    int do_visina = *max_element(begin(visine), end(visine));
    while (od_visina <= do_visina) {
        int visina = od_visina + (do_visina - od_visina) / 2;
        int naseceno = 0;
        for (int v : visine)
            if (v >= visina)
                naseceno += v - visina;
        if (naseceno >= potrebno) od_visina = visina + 1;
        else  do_visina = visina - 1;
    }
    return do_visina;
}
```

Najveći broj porcija

Problem: Kuvar pravi jelo koje sadrži n raznih sastojaka. Od svakog sastojka u kuhinji već ima određeni broj grama. Dostupan mu je određeni iznos novca i za to može da kupuje mala ili velika pakovanja svakog od sastojka (poznata je cena i gramaža svakog malog i svakog velikog pakovanja). Napisati program koji određuje maksimalni broj porcija koje kuvar može da spremi.

Osnovna ideja je da problem optimizacije svedemo na problem odlučivanja tj. da napravimo funkciju koja proverava da li je dati broj porcija moguće napraviti sa iznosom dinara koji imamo na raspolaganju. Kada znamo broj potrebnih porcija možemo odrediti potrebnu količinu svakog sastojka. Pod pretpostavkom da na neki način umemo da izračunamo najmanju cenu za koju je moguće kupiti tu količinu sastojka, možemo izračunati zbir tih minimalnih cena za sve sastojke i dobiti minimalnu ukupnu cenu potrebnu da se kupe sastojci dovoljni da se napravi dati broj porcija. Dati broj porcija je moguće napraviti akko je ta ukupna minimalna cena manja ili jednaka iznosu novca koji imamo na raspolaganju.

Pretpostavljamo da znamo gramažu i cenu malog i gramažu i cenu velikog pakovanja. Ideja je da isprobamo sve moguće kombinacije brojeva malih i velikih pakovanja koje nam daju potrebnu gramažu. Krećemo od toga da kupujemo samo velika pakovanja. Broj potrebnih pakovanja izračunavamo zaokruživanjem celobrojnog količnika potrebne gramaže i gramaže velikog pakovanja naviše. Nakon toga smanjujem broj velikih pakovanja za po jedno i izračunavamo broj potrebnih malih pakovanja kojim se dopunjuje nedostajuća gramaža koju izračunavamo tako što od potrebne gramaže oduzmemo proizvod broja velikih pakovanja i gramaže velikog pakovanja. Broj malih pakovanja izračunavamo opet zaokruživanjem količnika te preostale gramaže i gramaže malog pakovanja naviše. Kada znamo broj malih i velikih pakovanja lako izračunavamo njihovu cenu i ako je manja od do tada minimalne, ažuriramo minimum.

Kada imamo funkciju provere, optimimum tražimo binarnom pretragu. Gornju granicu možemo grubo proceniti i na osnovu ulaznih parametara. Još jednostavniji način je da primenimo tehniku binarne pretrage u kojoj gornju granicu duplo povećavamo sve dok ne dobijemo previše porcija. Kada imamo donju i gornju granicu, binarna pretraga teče na uobičajeni način.

```
int brojSastojaka, dostupanIznosNovca;
vector<int> potrebnoGrama, dostupnoGrama, maloGrama, maloCena;
vector<int> velikoGrama, velikoCena;

int mozeSeNapraviti(int brojPorcija) {
    int ukupnaCena = 0;
    for (int i = 0; i < n; i++) {
        int potrebno = brojPorcija*potrebnoGrama[i]-postojiGrama[i];
        ukupnaCena += minCenaZaKolicinu(malaCena[i], malaKolicina[i],
                      velikaCena[i], velikaKolicina[i], potrebno);
    }
    return ukupnaCena <= dostupanIznosNovca;
}
```

```
int ceilFrac(int a, int b) {
    return (a + b - 1) / b;
}

int minCena(int maloCena, int maloGrama, int velikoCena,
           int velikoGrama, int potrebnoGrama) {
    int najviseVelikih = ceilFrac(potrebnoGrama, velikoGrama);
    int min = najviseVelikih * velikaCena;
    for (int velikih = najviseVelikih - 1; velikih >= 0; velikih--) {
        int potrebnaKolicinaMalih = potrebnaKolicina
                - velikih * velikaKolicina;
        int malih = ceilFrac(potrebnaKolicinaMalih, malaKolicina);
        int cena = velikih * velikaCena + malih * malaCena;
        if (cena < min)
            min = cena;
    }
    return min;
}

int l = 0, d = 1;
while (mozeSeNapraviti(d))
    d *= 2;
while (l <= d) {
    int s = l + (d - l) / 2;
    if (mozeSeNapraviti(s))
        l = s + 1;
    else
        d = s - 1;
}
cout << d << endl;
```

7 Tehnika dva pokazivača

7.1 Dva pokazivača - objedinjavanje dva sortirana niza

Problem: Data su dva sortirana niza. Kreirati treći sortirani niz koji sadrži tačno sve elemente prethodna dva onoliko puta koliko se ukupno javljaju u oba niza.
Standardna biblioteka jezika C++ sadrži funkciju *merge*. Ako se elementi u nizovima ne ponavljaju, onda su tim nizovima predstavljeni skupovi, a objedinjavanje gradi skupovnu uniju. U jeziku C++ postoje i funkcije *set_union*, *set_interesection*, $set\_difference$ i $set\_symmetric\_difference$ koje određuju skupovne operacije nad skupovima predstavljenim sortiranim nizovima.

```
void merge(int a[], int na, int b[], int nb, int c[]) {
    int i = 0, j = 0, k = 0;
    while (i < na && j < nb)
        c[k++] = a[i] < b[j] ? a[i++] : b[j++];
    while (i < na)
        c[k++] = a[i++];
    while (j < nb)
        c[k++] = a[j++];
}
```

7.2 Dva pokazivača - par brojeva datog zbira, par brojeva date razlike, trojka datog zbira

Par brojeva datog zbira

Problem: Definisati algoritam složenosti *O(n)* koji određuje koliko u datom rastuće sortiranom nizu dužine n (svi elementi su različiti) postoji parova brojeva na različitim pozicijama čiji je zbir jednak datom broju s.
Pošto je niz brojeva sortiran, jedan način da se zadatak reši je da se za svaki element a_i proveri da li se u delu iza njega javlja element $s - a_i$, što može biti urađeno binarnom pretragom. Ovim bi se dobio algoritam složenosti *O(nlogn)*, međutim, može se i efikasnije od toga.

```
int brojParova(const vector<int>& a, int s, int l, int d) {
    if (l >= d)
        return 0;
    if (a[l] + a[d] > s)
        return brojParova(a, s, l, d - 1);
    else if (a[l] + a[d] < s)
        return brojParova(a, s, l + 1, d);
    else
        return 1 + brojParova(a, s, l + 1, d - 1);
}
```

Par brojeva date razlike

Problem: Definisati algoritam složenosti *O(n)* koji određuje koliko parova elemenata na različitim pozicijama u nizu imaju razliku jednaku datom broju r>0. Naivan način da se zadatak reši je da se ispitaju svi parovi brojeva i da se prebroje oni čija je razlika jednaka traženoj, $O(n^2)$.

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

int brojParovaDateRazlike(vector<int>& a, int razlika) {
    sort(begin(a), end(a));
    return brojParovaDateRazlike(a, razlika, 0, 1);
}
```

Trojka brojeva sa zbirom nula

Problem: Definisati algoritam složenosti O(n 2) koji određuje da li u datom sortiranom nizu dužine n postoji trojka elemenata čiji je zbir 0.
Naivno rešenje je da se provere sve trojke elemenata za i < j < k, tj. da se upotrebe tri ugnežđene petlje u čijem se telu proverava da li je ispunjen uslov i ako jeste, da se uveća brojač. Složenost ovog algoritma odgovara broju trojki i jednaka je $O(n^3)$. Pretpostavimo da se elementi u trojkama javljaju u istom redosledu u kom se javljaju u nizu. Da bi se neka trojka čiji je zbir 0 mogla formirati, potrebno je da se u delu niza iza elementa a_i nađe par elemenata (a_j, a_k) takav da je zbir ta dva elementa jednak -a_i. Pošto je ceo niz sortiran, svaki sufiks iza pozicije i će biti sortiran, tako da na njega možemo primeniti algoritam obilaska niza sa dva pokazivača koji smo objasnili prilikom brojanja parova datog zbira. Pošto je taj algoritam složenosti *O(n)*, ukupna složenost je $O(n^2)$.

```
int trojkaZbiraNula(const vector<int>& a) {
    int brojTrojki = 0;
    for (int i = 0; i < n - 2; i++) {
        int l = i + 1;
        int d = n - 1;
        while (l < d) {
            if (a[i] + a[l] + a[d] > 0)
                d--;
            else if (a[i] + a[l] + a[d] < 0)
                l++;
```

```
            else {
                brojTrojki++;
                l++;
                d--;
            }
        }
    }
    return brojTrojki;
}
```

7.3 Dva pokazivača - pretraga dvostruko sortirane matrice

Problem: Svaka vrsta i svaka kolona matrice dimenzije $m \quad n$ je sortirana neopadajuće. Napisati program koji u složenosti *O(m+n)* proverava da li matrica sadrži neki dati element. Prilagodi algoritam tako da u istoj složenosti određuje broj pojavljivanja datog elementa.
Naivni način da zadatak rešimo je da primenimo linearnu pretragu kroz sve elemente matrice. Složenost najgoreg slučaja nastupa kada elementa nema u matrici i iznosi *O(m·n)*. Binarnom pretragom možemo postići da se svaka vrsta dužine n pretraži u *O(logn)* koraka, pa je ukupna složenost *O(mlogn)*. Ako je $m > n$, tada je bolje binarnu pretragu primenjivati na kolone. I ovo rešenje se može pojednostaviti primenom bibliotečkih funkcija jezika C++.
Zahvaljujući dvostrukoj sortiranosti, takođe je sortiran svaki zglob matrice koji se sastoji od bilo kog pocetka neke kolone i ostatka vrste od poslednjeg elementa tog pocetka kolone. Eliminacijom jedne strane zgloba, eliminise se jedna kolona ili jedna vrsta matrice. Potom se ovaj postupak rekurzivno ponavlja. Jasno je da je broj poređenja najviše $m + n$, pa da je složenost *O(m+n)*.

```
bool sadrzi(int a[MAX][MAX], int m, int n, int v, int k, int x) {
    if (v < 0 || k >= n)
        return false;
    if (a[v][k] < x)
        return sadrzi(a, m, n, v, k+1, x);
    else if (a[v][k] > x)
        return sadrzi(a, m, n, v-1, k, x);
    else
        return true;
}
```

7.4 Dva pokazivača - segmenti niza prirodnih brojeva datog zbira

Problem: U datom nizu pozitivnih prirodnih brojeva naći sve segmente (njihov početak i kraj) čiji je zbir jednak datom pozitivnom broju (brojanje pozicija počinje od nule).
Obeležimo sa $z_{ij} = \sum_{k=i}^{j} a_k$ zbir elemenata niza a čiji indeksi pripadaju segmentu [i, j], a sa z traženi zbir elemenata. Pošto su svi elementi niza a pozitivni, zbirovi elemenata segmenta zadovoljavaju svojstvo monotonosti. Pretpostavimo da za neki interval [i, j] znamo da za svako *j'* takvo da je i $\leq$ j' $<$ j važi da je $z_{ij'} < z$. Postoje sledeći slučajevi za odnos z_{ij} i z:

• Prvo, ako je $z_{ij} < z$ tada ni za jedan interval koji počinje na poziciji i, a završava se najkasnije na poziciji j ne može važiti da mu je zbir elemenata z, i proveru je potrebno nastaviti od intervala [i, j + 1], uvećavajući j za 1. Ako takav interval ne postoji (ako je j + 1 = n), onda se pretraga može završiti.

• Drugo, pretpostavimo da je $z_{ij} \geq z$. Ako je $z_{ij} = z$, tada je pronađen jedan zadovoljavajući interval i potrebno je obraditi njegove granice i i j. To može biti jedini segment koji počinje na poziciji i sa zbirom z. Pošto su svi elementi niza a pozitivni, za svako *j"* takvo da je j $<$ j" $<$ n važi da je $z \leq z_{ij} < z_{ij''}$. Dakle, pretragu možemo nastaviti uvećavajući vrednost i. Za sve vrednosti *j'* takve da je i+1 $\leq$ j' $<$ j važi da je $z_{(i+1)j'} < z$. Pošto je $a_i > 0$ važi da je $z_{(i+1)j'} < z_{ij'} < z$. Dakle, na segment [i+1, j] može se primeniti analiza slučajeva istog oblika kao na interval [i, j].

Prilikom implementacije održavaćemo interval [i, j] i njegov zbir ćemo izračunavati inkrementalno - prilikom povećanja broja j zbir ćemo uvećavati za a_j, a prilikom povećanja broja i zbir ćemo umanjivati za a_i.

```
void ispisiSegmenteZbira(const vector<int>& a, int trazeniZbir) {
    int i = 0, j = 0;
    int zbir = a[0];
    while(true) {
        if (zbir < trazeniZbir) {
            j++;
            if (j >= n)
                break;
            zbir += a[j];
        }
        else {
            if (zbir == trazeniZbir)
                cout << i << " " << j << endl;
            zbir -= a[i];
            i++;
        }
    }
}
```

7.5 Dva pokazivača - najkraća podniska koja sadrži sva data slova

Problem: Definisati funkciju koja u datoj niski određuje najkraću podnisku koja sadrži sve karaktere iz datog skupa karaktera.
Najdirektniji algoritam bio bi da se razmatraju sve podniske, da se za svaku podnisku proveri da li je ispravna tj. da li sadrži sve karaktere iz skupa S i da se među svim ispravnim pronađe najkraći. Ovakvu pretragu grubom silom je veoma jednostavno implementirati, ali njena složenost je $O(n^3m)$ gde je n dužina teksta, a m broj karaktera u skupu S.
Može se primetiti da kada segment određen pozicijama i i j sadrži sve karaktere iz skupa S, onda i svi segmenti određeni većim vrednostima j takođe sadrže sve karaktere iz skupa, a za njih znamo da su duži i nema potrebe razmatrati ih. Prva pronađena ispravna podniska koja počinje na poziciji i ujedno je i najkraća ispravna podniska pronađena na toj poziciji. Zato je odmah nakon pronalaska prve ispravne podniske i ažuriranja vrednosti najmanje dužine moguće prekinuti unutrašnju petlju. Ova optimizacija ne popravlja asimptotsku složenost najgoreg slučaja.
Primetimo da najkraća podniska mora da počinje i da se završava karakterom iz skupa S. U suprotnom bi karakteri na početku i na kraju podteksta koji nisu u skupu S mogli da budu uklonjeni čime bi se dobila kraća podniska koja bi i dalje sadržala sve karaktere iz skupa S. Dakle, prilikom razmatranja podniski dovoljno je razmatrati samo njihove karaktere iz skupa S, pa ćemo u prvoj fazi samo izgraditi niz njihovih pozicija u niski. Tada je potrebno razmatrati samo segmente koji počinju i završavaju se na pozicijama unutar tog vektora, što u slučaju da postoji značajan broj karaktera u tekstu koji nisu u skupu S može ubrzati pretragu. Vreme potrebno za izgradnju niza relevantnih pozicija je $O(nm)$.
Proveru da li se svi karakteri iz skupa S javljaju u segmentu između dve relevantne pozicije i i j možemo izvršiti tako što odredimo skup svih karaktera na svim relevantnim pozicijama između i i j. Svi karakteri iz tako napravljenog skupa će biti u skupu S, jer razmatramo samo relevantne pozicije tj. samo pozicije na kojima smo prethodno ustanovili da se nalaze karakteri iz S, pa je umesto ispitivanja jednakosti dva skupa, dovoljno ustanoviti samo da li imaju isti broj elemenata. Izgradnju skupa karaktera koji su se pojavili u tekućem segmentu možemo vršiti inkrementalno, tako što prilikom svakog povećanja j, tj. prilikom prelaska na novu relevantnu poziciju j dodamo karakter na toj poziciji u taj skup, što zahteva samo konstantno vreme. Složenost ovog pristupa ako i dalje vršimo ispitivanje svih početnih pozicija je $O(n^2)$.
Bolji algoritam možemo konstruisati tako što i dalje obrađujemo sve relevantne pozicije redom, sa leva na desno, ali za svaku od njih, umesto najkraće podniske koja na njoj počinje, pronalazimo najkraću podnisku koja se na njoj završava. Ako za svaku relevantnu poziciju znamo najkraću podnisku koji se na njoj završava, od svih takvih možemo naći najkraću i ona će ujedno biti i globalno najkraća.

Najkraća podniska će biti pronađena jer se ona sigurno završava na nekoj relevatnoj poziciji i ujedno je najkraća od svih koji se na toj poziciji završavaju, tako da će sigurno biti uzeta u obzir prilikom određivanja najmanje dužine. Uvek ćemo znati najkraću ispravnu podnisku koji se završava na prethodnoj relevatnoj poziciji, kao i to da ako se prvi relevantan karakter u podniski javlja bar još jednom kasnije u podniski, onda ta podniska ne može biti najkraća koja sadrži sve karaktere skupa S.
Algoritam će prvo pronaći prvu ispravnu podnisku (ako takva postoji). Zatim će se obrađivati jedna po jedna relevantna pozicija nadesno, i za nju će se određivati najkraća ispravna podniska koja se na njoj završava, tako što će se kretati od najkraće ispravne podniske koja se završava na prethodnoj relevantnoj poziciji, zatim će se on proširivati nadesno da uključi relevantne karaktere do trenutne relevantne pozicije i zatim će se iz te podniske izbacivati prefiksi, sve dok se ne naiđe na relevantni karakter koji se ne javlja kasnije u podniski.
Trenutna podniska biće određena pomoću dva iteratora i i j niza relevantnih pozicija, a pošto za svaki karakter treba da znamo koliko puta se javlja u podniski, koristimo mapu koja svaki karakter preslikava u njegov broj pojavljivanja.
Spoljna petlja po j prolazi kroz sve relevantne pozicije kojih ima najviše $O(n)$.
Unutrašnja petlja se može izvršiti ukupno $O(n)$ puta. U okviru petlji vrši se ažuriranje vrednosti u mapi, ali s obzirom na to da ta operacija ima logaritamsku složenost, a broj karaktera u skupu S je mali, možemo reći da je složenost algoritma $O(n)$.

```
int najkracaPodniska(const string& niska, const string& S) {
    vector<int> poz_karaktera_iz_S;
    for(int i = 0; i < niska.size(); i++)
        if(S.find(niska[i]) != string::npos)
            poz_karaktera_iz_S.push_back(i);
    int min_duzina = numeric_limits<int>::max();
    map<char, int> broj_pojavljivanja_u_podniski;
    vector<int>::const_iterator i, j;
    for(i = j = poz_karaktera_iz_S.begin(); j !=
         poz_karaktera_iz_S.end(); j++){
        broj_pojavljivanja_u_podniski[niska[*j]]++;
        if (broj_pojavljivanja_u_podniski.size() == S.size()) {
            while (broj_pojavljivanja_u_podniski[niska[*i]] > 1) {
                broj_pojavljivanja_u_podniski[niska[*i]]--;
                i++;
            }
            int duzina = *j - *i + 1;
            if (duzina < min_duzina) min_duzina = duzina;
        }
    }
    if (min_duzina != numeric_limits<int>::max())
        return min_duzina;
    else  return -1;
}
```

8 Dekompozicija

8.1 Tehnika dekompozicije - definicija, složenost

U mnogim situacijama efikasni algoritmi se mogu dobiti time što se niz podeli na dva dela koji se nezavisno obrađuju i nakon toga se konačni rezultat dobija objedinjavanjem tako dobijenih rezultata. Ova tehnika se naziva *tehnika razlaganja*, *tehnika dekompozicije* ili *tehnika podeli-pa-vladaj* (engl. divide- and-conquer).
Ako su delovi koji se obrađuju jednaki, dobija se jednačina T(n) = 2T(n/2) + O(n), T(0) = O(1), čije je rešenje *O(nlogn)*, jednačina T(n) = 2T(n/2) + O(1), T(0) = O(1) čije je rešenje *O(n)* ili jednačina T(n) = 2T(n/2) + O(logn), T(0) = O(1) čije je rešenje *O(n)*.
Treba obratiti pažnju na to da ako su polovine neravnomerne, moguće je da se dobije proces koji se opisuje jednačinom T(n) = T(n-1) + O(n), T(0) = O(1), čije je rešenje $O(n^2)$.
U nekim slučajevima nije neophodno rešavati oba potproblema. Tipičan primer je algoritam binarne pretrage. Jednačina koja se u tom slučaju dobija je T(n) = T(n/2) + O(1), T(0) = O(1) čije je rešenje *O(logn)* ili T(n) = T(n/2) + O(n), T(0) = O(1), čije je rešenje *O(n)*.

8.2 Dekompozicija - sortiranje objedinjavanjem (Merge sort)

Problem: Implementirati sortiranje niza objedinjavanjem sortiranih polovina.

```
void merge(vector<int>& a, int i, int m, vector<int>& b, int j,
int n, vector<int>& c, int k) {
    while (i <= m && j <= n)
        c[k++] = a[i] <= b[j] ? a[i++] : b[j++];
    while (i <= m)
        c[k++] = a[i++];
    while (j <= n)
        c[k++] = b[j++];
}

void merge_sort(vector<int>& a, int l, int d, vector<int>& tmp) {
    if (l < d) {
        int s = l + (d - l) / 2;
        merge_sort(a, l, s, tmp);
        merge_sort(a, s+1, d, tmp);

        merge(a, l, s, a, s+1, d, tmp, l);
        for (int i = l; i <= d; i++)
            a[i] = tmp[i];
    }
}
```

8.3 Dekompozicija - brojanje inverzija

Problem: Odredi koliko različitih parova elemenata u nizu je takvo da je prvi element strogo veći drugog.
Jedan način da se odredi broj inverzija je da se niz sortira sortiranjem objedinjavanjem, prilagođenim tako da se broje inverzije. Rekurzivno određujemo broj inverzija u levoj i desnoj polovini niza. Nakon toga, prilikom objedinjavanja određujemo broj inverzija tako da je prvi element u levoj, a drugi u desnoj polovini niza. Primenjujemo klasičan algoritam objedinjavanja (zasnovan na tehnici dva pokazivača) i ako se dogodi da je tekući element u desnoj polovini niza strogo manji od tekućeg elementa u levoj polovini niza, znamo da je strogo manji i od svih elemenata u levoj polovini iza njega. Znači taj element učestvuje u onoliko inverzija koliko je još ostalo elemenata u levoj polovini niza. Pošto se niz tokom algoritma sortira, ako želimo da zadržimo njegov originalni sadržaj, potrebno ga je pre primene algoritma prekopirati u pomoćni niz.

```
int broj_inverzija(vector<int>& a, int l, int d, vector<int>& b) {
    if (l >= d)
        return 0;
    int s = l + (d - l) / 2;
    int broj = 0;
    broj += broj_inverzija(a, l, s, b);
    broj += broj_inverzija(a, s+1, d, b);
    int pl = l, pd = s+1, pb = 0;

    while (pl <= s && pd <= d) {
        if (a[pl] <= a[pd])
            b[pb++] = a[pl++];
        else {
            broj += s - pl + 1;
            b[pb++] = a[pd++];
        }
    }
    while (pl <= s)
        b[pb++] = a[pl++];
    while (pd <= d)
        b[pb++] = a[pd++];

    for (int i = l; i <= d; i++)
        a[i] = b[i-l];
    return broj;
}

int broj_inverzija(const vector<int>& a) {
    vector<int> pom1(a.size()), pom2(a.size());
    for (int i = 0; i < a.size(); i++)
        pom1[i] = a[i];
    return broj_inverzija(pom1, 0, pom1.size()-1, pom2);
}
```

8.4 Dekompozicija - brzo sortiranje (Quick sort)

Problem: Sortirati niz brojeva primenom algoritma brzog sortiranja.

```
void quick_sort(vector<int>& a, int l, int d) {
    if (l < d) {
        swap(a[l], a[l + rand() % (d - l + 1)]);
        int k = l;
        for (int i = l+1; i <= d; i++)
            if (a[i] <= a[l])
                swap(a[i], a[++k]);
        swap(a[l], a[k]);
        quick_sort(a, l, k - 1);
        quick_sort(a, k + 1, d);
    }
}
```

8.5 Dekompozicija - k najvećih elemenata (Quick select)

Problem: U nizu od n elemenata pronaći element od kojega je tačno k elemenata manje ili jednako.
Jedno rešenje je da se ceo niz sortira i da se onda vrati element na poziciji k. Međutim, ovo može biti neefikasno. Zamislimo, da je k mali broj, na primer 0. Tada se traži minimim niza, a rešenje zasnovano na sortiranju nepotrebno određuje međusobni odnos svih elemenata iza njega. Umesto algoritma složenosti *O(n)* koristimo algoritam složenosti *O(nlogn)*.
Rešenje koje je efikasnije od sortiranja se zasniva na modifikaciji algoritma QuickSort koja je poznata pod imenom QuickSelect. Pokažimo varijantu koja određuje element niza na poziciji k.
Isto kao i u slučaju algoritma QuickSort, algoritam QuickSelect počinje odabirom pivotirajućeg elementa i particionisanjem niza. Nakon particionisanja postoje tri mogućnosti. Neka je pozicija pivota nakon particionisanja p. Jedna je da se pivot nalazi baš na traženoj poziciji n, tj. da je $p = k$ i u tom slučaju funkcija vraća pivot i završava sa radom. Druga mogućnost je da je $k < p$ i tada se algoritam primenjuje na deo niza [l, p-1]. Na kraju, treća mogućnost je da je $k > p$ i tada se algoritam primenjuje na deo niza [p+1, d]. Osnovna implementacija može biti rekurzivna, međutim, pošto se u varijanti QuickSelect vrši samo jedan rekurzivni poziv i to kao repni, rekurziju je veoma jednostavno eliminisati.
Složenost algoritma QuickSelect zavisi od toga koliko sreće imamo da pivot ravnomerno podeli interval [l, d]. Ako bi se u svakom koraku desilo da pivot upadne na sredinu intervala, ukupan broj poređenja i razmena bio bi *O(n)*. Ako bi pivot stalno bio blizak nekom od dva kraja intervala [l, d], tada bi složenost algoritma bila $O(n^2)$. Prosečna složenost algoritma QuickSelect jednaka je *O(n)*.

```
nth_element(a.begin(), next(a.begin(), k), a.end(), greater<int>());
```

```
int ktiElement(vector<int>& a, int l, int d, int k) {
    while (true) {
        swap(a[l], a[random_value(l, d)]);
        int i = l + 1, j = d;
        while (i <= j) {
            if (a[i] < a[l])
                i++;
            else if (a[j] > a[l])
                j--;
            else
                swap(a[i++], a[j--]);
        }
        swap(a[l], a[j]);
        if (k < j)
            d = j - 1;
        else if (k > j)
            l = j + 1;
        else
            return a[k];
    }
}
```

8.6 Dekompozicija - maksimalni zbir segmenta

Problem: Definisati efikasnu funkciju koja pronalazi najveći mogući zbir segmenta (podniza uzastopnih elemenata) datog niza brojeva.
Dekompozicija nam sugeriše da je poželjno da niz podelimo na dva podniza jednake dužine čija rešenja možemo da konstruišemo na osnovu induktivne hipoteze. Bazu čini slučaj praznog niza, koji sadrži samo prazan segment čiji je zbir nula. Fiksirajmo središnji element niza. Sve segmente niza možemo da grupišemo u tri grupe: segmente koji su u potpunosti levo od središnjeg elementa, segmente koji su u potpunosti desno od središnjeg elementa i segmente koji sadrže središnji element. Najveće zbirove segmenata u prvoj i u drugoj grupi znamo na osnovu induktivne hipoteze. Najveći zbir segmenta u trećoj grupi možemo lako odrediti analizom svih segmenata: krećemo od jednočlanog segmenta koji sadrži samo središnji element i inkrementalno se širimo nalevo dodajući jedan po jedan element i računajući tekući maksimum, a zatim krećemo od maksimalnog segmenta proširenog nalevo i inkrementalno ga proširujemo jednim po jednim elementom nadesno, tražeći novi maksimum.
Ako sa n označimo dužinu niza d-l+1 i ako vreme izvršavanja obeležimo sa $T(n)$, tada važi da je $T(0) = O(1)$ i da je $T(n) = 2T(n/2) + O(n)$. Vrše se dva rekurzivna poziva za duplo manje nizove, a najveći zbir segmenata koji obuhvataju središnji element izračunavamo u vremenu $O(n)$. Na osnovu master teoreme lako se zaključuje da je $T(n) = O(n \log n)$.

```
int maksZbirSegmenta(int a[], int l, int d) {
    if (l > d)
        return 0;
    int s = l + (d - l) / 2;
    int maks_zbir_levo = maksZbirSegmenta(a, l, s-1);
    int maks_zbir_desno = maksZbirSegmenta(a, s+1, d);
    int zbir_sredina = a[s];
    int maks_zbir_sredina = zbir_sredina;
    for (int i = s-1; i >= l; i--) {
        zbir_sredina += a[i];
        if (zbir_sredina > maks_zbir_sredina)
            maks_zbir_sredina = zbir_sredina;
    }
    zbir_sredina = maks_zbir_sredina;
    for (int i = s+1; i <= d; i++) {
        zbir_sredina += a[i];
        if (zbir_sredina > maks_zbir_sredina)
            maks_zbir_sredina = zbir_sredina;
    }
    return max({maks_zbir_levo, maks_zbir_desno, maks_zbir_sredina});
}
```

Na ideji dekompozicije možemo izgraditi i efikasniji algoritam. Najveći zbir segmenta oko srednjeg elementa se može dobiti kao zbir najvećeg sufiksa niza levo od tog elementa i najvećeg prefiksa niza desno od tog elementa. Možemo ojačati induktivnu hipotezu i umesto da prefiks i sufiks računamo u petlji, u linearnom vremenu, možemo pretpostaviti da za obe polovine niza prefiks i sufiks dobijamo kao rezultat rekurzivnog poziva. To nam je dovoljno da odredimo maksimalni zbir funkcije i naša funkcija sada pored maksimalnog zbira segmenta mora izračunati i maksimalni zbir prefiksa i maksimalni zbir sufiksa celog niza. Maksimalni zbir prefiksa celog niza je veći broj od maksimalnog zbira prefiksa levog dela i od zbira celog levog dela i maksimalnog zbira prefiksa desnog dela. Slično, maksimalni zbir sufiksa celog niza je veći od maksimalnog zbira sufiksa desnog dela i od zbira maksimalnog zbira sufiksa levog dela i celog desnog dela. Zato je neophodno dodatno ojačati induktivnu hipotezu i tokom rekurzije računati i zbir celog niza. Složenost ovakve implementacije zadovoljava jednačinu $\mathrm{T(n)} = 2\mathrm{T(n/2)}+\mathrm{O(1)}$ i jednaka je *O(n)*.

```
void maksZbirSegmenta(const vector<int>& a, int l, int d,
int& zbir, int& maks_zbir, int& maks_prefiks, int& maks_sufiks) {
    if (l == d) {
        zbir = maks_zbir = maks_prefiks = maks_sufiks = a[l];
        return;
    }
    int s = l + (d - l) / 2;

    int zbir_levo, maks_zbir_levo;
    int maks_sufiks_levo, maks_prefiks_levo;
```

```
    maksZbirSegmenta(a, l, s, zbir_levo, maks_zbir_levo,
                  maks_prefiks_levo, maks_sufiks_levo);

    int zbir_desno, maks_zbir_desno;
    int maks_sufiks_desno, maks_prefiks_desno;
    maksZbirSegmenta(a, s+1, d,zbir_desno, maks_zbir_desno,
                  maks_prefiks_desno, maks_sufiks_desno);

    zbir = zbir_levo + zbir_desno;
    maks_prefiks = max(maks_prefiks_levo, zbir_levo +
                     maks_prefiks_desno);
    maks_sufiks = max(maks_sufiks_desno, maks_sufiks_levo +
                    zbir_desno);
    maks_zbir = max({maks_zbir_levo, maks_zbir_desno,
                   maks_sufiks_levo + maks_prefiks_desno});
}

int maksZbirSegmenta(const vector<int>& a) {
    int zbir, maks_zbir, maks_prefiks, maks_sufiks;
    maksZbirSegmenta(a, 0, a.size() - 1, zbir, maks_zbir,
                   maks_prefiks, maks_sufiks);
    return maks_zbir;
}
```

8.7 Dekompozicija - silueta zgrada

Problem: Sa broda se vide zgrade na obali velegrada. Duž obale je postavljena koordinatna osa i za svaku zgradu se zna pozicija levog kraja, visina i pozicija desnog kraja. Napisati program koji izračunava siluetu grada.
Svaku zgradu predstavljaćemo strukturom koja sadrži levi kraj zgrade a, zatim desni kraj zgrade b i njenu visinu h.

```
struct zgrada {
    int a, b, h;
    zgrada(int a = 0, int b = 0, int h = 0) : a(a), b(b), h(h) {
    }
};
```

Svaki uređeni par (x_i, h_i) predstavljaćemo strukturom *promena*, a siluetu ćemo predstavljati vektorom *promena*.

```
struct promena {
    int x, h;
    promena(int x = 0, int h = 0) : x(x), h(h) {
    }
};

vector<promena> silueta;
```

Direktan induktivno-rekurzivni pristup bi podrazumevao da se prvo napravi silueta svih zgrada osim poslednje i da se onda ažurira na osnovu poslednje zgrade (bazni slučaj je silueta jedne zgrade). Vreme potrebno da se zgrada umetne u siluetu zavisi od dužine siluete koja je u najgorem slučaju jednaka broju obrađenih zgrada. Ovim se dobija jednačina T(n) = T(n-1) + O(n), T(1) = O(1), čije je rešenje $O(n^2)$. Problem možemo efikasnije rešiti tehnikom podeli-pa-vladaj. Dve siluete možemo objediniti za isto vreme za koje možemo objediniti jednu zgradu u siluetu. Pošto su siluete sortirane možemo ih obilaziti uz održavanje dva pokazivača i objedinjavati veoma slično objedinjavanju dva sortirana niza brojeva.

```
vector<promena> silueta(const vector<zgrada>& zgrade, int l, int d) {
    vector<promena> rezultat;
    if (l == d) {
        rezultat.emplace_back(zgrade[l].a, zgrade[l].h);
        rezultat.emplace_back(zgrade[l].b, 0);
        return rezultat;
    }
    int s = l + (d - l) / 2;
    vector<promena> rezultat_l = silueta(zgrade, l, s);
    vector<promena> rezultat_d = silueta(zgrade, s+1, d);
    int ll = 0, dd = 0;
    int Hl = 0, Hd = 0;
    while(ll < rezultat_l.size() || dd < rezultat_d.size()) {
        int x;
        if (ll == rezultat_l.size()) {
            x = rezultat_d[dd].x;
            Hd = rezultat_d[dd].h;
            dd++;
        }
        else if(dd == rezultat_d.size()) {
            x = rezultat_l[ll].x;
            Hl = rezultat_l[ll].h;
            ll++;
        }
        else {
            int xl = rezultat_l[ll].x;
            int xd = rezultat_d[dd].x;
            if (xl <= xd) {
                x = xl;
                Hl = rezultat_l[ll].h;
                ll++;
            }
            else {
                x = xd;
                Hd = rezultat_d[dd].h;
                dd++;
            }
        }
```

```
            int h = max(Hl, Hd);
            if (rezultat.size() > 0) {
                int xb = rezultat.back().x, hb = rezultat.back().h;
                if (x == xb)
                    rezultat.back().h = h;
                else if (h != hb)
                    rezultat.emplace_back(x, h);
            }
            else
                rezultat.emplace_back(x, h);
        }
        return rezultat;
    }
```

8.8 Dekompozicija - Karacubin algoritam

Problem: Definisati funkciju koja množi dva polinoma predstavljena vektorima svojih koeficijenata. Jednostavnosti radi pretpostaviti da su vektori dužine 2^k. Klasičan algoritam množenja ima složenost $O(n^2)$.
Anatolij Karacuba je 1960. pokazao da je dekompozicijom moguće dobiti efikasniji algoritam. Pretpostavimo da je potrebno pomnožiti polinome $a+bx$ i $c+dx$. Direktan pristup podrazumeva izračunavanje $ac+(ad+bc)x+bd$, što podrazumeva 4 množenja. Karacubina ključna opaska je da se isto može ostvariti samo sa tri množenja (na račun malo većeg broja sabiranja tj. oduzimanja). Naime, važi da je $ad+bc = (a+b)(c+d) - (ac+bd)$. Potrebno je, dakle, samo izračunati proizvode ac, bd i $(a+b)(c+d)$, a onda prva dva proizvoda upotrebiti po dva puta.

```
vector<double> karacuba(const vector<double>& p1, const
vector<double>& p2) {
    int n = p1.size();
    if (n == 1)
        return vector<double(1, p1[0] * p2[0])>;

    vector<double> a(n / 2), b(n / 2);
    copy_n(begin(p1), n/2, begin(a));
    copy_n(next(begin(p1), n/2), n/2, begin(b));

    vector<double> c(n / 2), d(n / 2);
    copy_n(begin(p2), n/2, begin(c));
    copy_n(next(begin(p2), n/2), n/2, begin(d));

    vector<double> ac = karacuba(a, c);
    vector<double> bd = karacuba(b, d);

    for (int i = 0; i < n/2; i++) a[i] += b[i];
    for (int i = 0; i < n/2; i++) c[i] += d[i];
```

```
    vector<double> adbc = karacuba(a, c);
    for (int i = 0; i < n; i++)
        adbc[i] -= ac[i] + bd[i];

    vector<double> proizvod(2*n, 0.0);
    for (int i = 0; i < n; i++) {
        proizvod[n + i] += bd[i];
        proizvod[n/2 + i] += adbc[i];
        proizvod[i] += ac[i];
    }
    return proizvod;
}
```

Složenost algoritma je određena rekurentnom jednačinom T(n) = 3T(n/2) + O(n) i iznosi $O(n^{log_3 2})$.
Međutim, prethodna implementacija je neefikasna i ne doprinosi poboljšanju efikasnosti naivne procedure. Ključni problem je to što se tokom rekurzije grade vektori u kojima se čuvaju privremeni rezultati i te alokacije i dealokacije troše jako puno vremena. Moguće je svu pomoćnu memoriju alocirati samo jednom i onda tokom rekurzije koristiti stalno isti pomoćni memorijski prostor. Veličina potrebne pomoćne memorije je $4n$. Dodatna optimizacija je da se primeti da je za male stepene polinoma klasičan algoritam brži nego algoritam zasnovan na dekompoziciji.

```
void karacuba(int n, const vector<double>& p1, int start1,
const vector<double>& p2, int start2, vector<double>& proizvod,
int start_proizvod, vector<double>& pom, int start_pom) {
    if (n <= 4) {
        for (int i = 0; i < 2*n; i++)
            proizvod[start_proizvod + i] = 0;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                proizvod[start_proizvod+i+j] += p1[start1+i] *
                    p2[start2+j];
        return;
    }

    karacuba(n/2, p1, start1, p2, start2, proizvod, start_proizvod,
        pom, start_pom);
    karacuba(n/2, p1, start1+n/2, p2, start2+n/2, proizvod,
        start_proizvod+n, pom, start_pom);

    for(int i = 0; i < n/2; i++)
        pom[start_pom+i] = p1[start1+i] + p1[start1+n/2+i];
    for(int i = 0; i < n/2; i++)
        pom[start_pom+n/2+i] = p2[start2+i] + p2[start2+n/2+i];

    karacuba(n/2, pom, start_pom, pom, start_pom+n/2, pom,
        start_pom+n, pom, start_pom+2*n);
```

```
    for(int i = 0; i < n; i++)
        pom[start_pom+n+i] -= proizvod[start_proizvod+i] +
            proizvod[start_proizvod+n+i];
    for(int i = 0; i < n; i++)
        pom[start_proizvod+n/2+i] += pom[start_pom+n+i];
}

vector<double> karacuba(const vector<double>& p1, const
     vector<double> p2) {
    int n = p1.size();
    vector<double> proizvod(2 * n);
    vector<double> pom(4 * n);
    karacuba(n, p1, 0, p2, 0, proizvod, 0, pom, 0);
    return proizvod;
}
```

8.9 Dekompozicija - najbliži par tačaka

Problem: U ravni je zadato n tačaka svojim koordinatama. Napiši program koji određuje najmanje (Euklidsko) rastojanje među njima.
Rešenje grubom silom podrazumeva ispitivanje svih parova tačaka i složenost mu je $O(n^2)$.
Jedan način da se do rešenja dođe efikasnije je da se primeni dekompozicija. Bazni slučaj predstavlja situacija u kojoj imamo manje od četiri tačke, jer njih ne možemo podeliti u dve polovine u kojima postoji bar po jedan par tačaka. U tom slučaju rešenje nalazimo poređenjem rastojanja svih parova tačaka (složenost je *O(1)*). Skup tačaka možemo jednom vertikalnom linijom podeliti na dve otprilike istobrojne polovine. Ako tačke sortiramo po koordinati x, vertikalna linija može odgovarati koordinati središnje tačke. Rekurzivno određujemo najmanje rastojanje u prvoj polovini (to su tačke levo od vertikalne linije) i u drugoj polovini (to su tačke desno od vertikalne linije). Najbliži par je takav da su: (1) obe tačke u levoj polovini, (2) obe tačke u desnoj polovini ili (3) jedna tačka je u levoj, a druga u desnoj polovini. Za prva dva slučaja već znamo rešenja i ostaje da se razmotri samo treći.
Neka je d_l minimalno rastojanje tačaka u levoj polovini, d_r minimalno rastojanje tačaka u desnoj polovini, a d manje od ta dva rastojanja. Ako vertikalna linija ima x-koordinatu x, tada je moguće odbaciti sve tačke koje su levo od x-d i desno od x+d, jer je njihovo rastojanje do najbliže tačke iz suprotne polovine sigurno veće od d. Potrebno je ispitati sve preostale tačke, tj. sve tačke iz pojasa [x-d, x+d], proveriti da li među njima postoji neki par tačaka čije je rastojanje strogo manje od d i vrednost d ažurirati na vrednost najmanjeg rastojanja takvog para tačaka. Problem je to što u najgorem slučaju njih može biti puno i ako ispitujemo sve parove, dolazimo u najgorem slučaju do oko $n^2/4$ poređenja. Ipak, proveru je moguće organizovati tako da se proveri samo mali broj parova tačaka.

Pretpostavimo da istovremeno razmatramo sve tačke unutar pojasa [x-d, x+d], bez obzira sa koje strane vertikalne linije se nalaze. Svaku tačku A iz pojasa je dovoljno uporediti sa onim tačkama koje leže unutar kruga sa centrom u tački A i poluprečnikom d, što omogućava značajna odsecanja. Pripadnost krugu nije jednostavno proveriti i zato umesto njega možemo razmatrati kvadrat stranice dužine $2d$ na čijoj se horizontalnoj srednjoj liniji nalazi tačka A. Time će odsecanje biti za nijansu manje nego u slučaju kruga, ali će detektovanje tačaka koje pripadaju tom pravougaoniku biti veoma jednostavno. To će biti sve one tačke iz pojasa kojima je koordinata y u intervalu [y_A-d, y_A+d]. Dalje smanjenje broja poređenja možemo dobiti ako primetimo da svaki par obrađujemo dva puta. Možemo jednostavno zaključiti da je dovoljno svaku tačku porediti samo sa onim tačkama koje se nalaze na istoj visini kao ona ili iznad nje. Dakle, svaku tačku je potrebno uporediti samo sa tačkama čije x koordinate leže unutar intervala [x-d, x+d] i čije y koordinate leže unutar intervala [y_A-d, y_A+d]. Prvi uslov možemo obezbediti tako što pre poređenja sve tačke iz pojasa širine d oko vertikalne linije podele izdvojimo u poseban niz ($O(n)$). Drugi uslov efikasnije možemo obezbediti ako sve tačke tog pomoćnog niza sortiramo po koordinati y ($O(nlogn)$) i zatim tačke obrađujemo u neopadajućem redosledu y koordinata. Za svaku tačku A obrađujemo samo tačke koje se nalaze iza nje u sortiranom nizu i obrađujemo jednu po jednu tačku sve dok ne naiđemo na tačku čija je koordinata y veća ili jednaka y_A+d.

```
bool porediX(const Tacka& t1, const Tacka& t2) {
    return t1.x <= t2.x;
}

bool porediY(const Tacka& t1, const Tacka& t2) {
    return t1.y <= t2.y;
}

double najblizeTacke(vector<Tacka>& tacke, int l, int r) {
    if (r - l + 1 < 4) {
        double d = numeric_limits<double>::max();
        for (int i = l; i < r; i++)
            for (int j = i+1; j <= r; j++) {
                double dij = rastojanje(tacke[i], tacke[j]);
                if (dij < d)
                    d = dij;
            }
        return d;
    }

    int s = l + (r - l) / 2;
    double d1 = najblizeTacke(tacke, l, s);
    double d2 = najblizeTacke(tacke, s+1, r);
    double d = min(d1, d2);
    double dl = tacke[s].x - d, dr = tacke[s].x + d;
    vector<Tacka> pojas;
```

```
    for (int i = l; i <= r; i++)
        if (dl <= tacke[i].x && tacke[i].x <= dr)
            pojas.push_back(tacke[i]);

    sort(begin(pojas), end(pojas), porediY);
    for (int i = 0; i < pojas.size(); i++)
        for (int j = i+1; j < pojas.size() && pojas[j].y - pojas[i].y
             < d; j++) {
            double dij = rastojanje(pojas[i], pojas[j]);
            if (dij < d)
                d = dij;
        }
    return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
    sort(begin(tacke), end(tacke), porediX);
    return najblizeTacke(tacke, 0, tacke.size() - 1);
}
```

Nakon rekurzivnih poziva, za dobijanje konačnog rezultata je potrebno izvršiti dodatnih $O(nlogn)$ koraka i dekompozicija zadovoljava rekurentnu jednačinu $\mathrm{T(n)} = 2\mathrm{T(n/2)} + \mathrm{O(nlogn)}$. Rešenje ove jednačine, na osnovu master teoreme, je $O(n(logn)^2)$.
Složenost se može popraviti ako se sortiranje po koordinati y vrši istovremeno sa pronalaženjem najbližeg para tačaka, tj. ako se ojača induktivna hipoteza i ako se pretpostavi da će rekurzivni poziv vratiti rastojanje između najbliže dve tačke i ujedno sortirati date tačke po koordinati y. U koraku objedinjavanja dva sortirana niza objedinjujemo u jedan. Na taj način dobijamo algoritam koji zadovoljava jednačinu $\mathrm{T(n)} = 2\mathrm{T(n/2)} + \mathrm{O(n)}$ i složenosti je $O(nlogn)$.

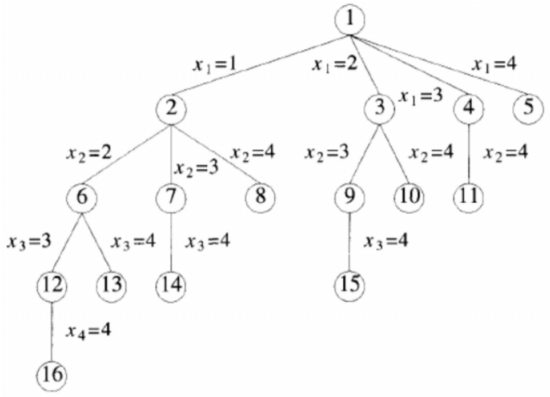
```
double najblizeTacke(vector<Tacka>& tacke, int l, int r,
vector<Tacka>& pom) {
    if (r - l + 1 < 4) {
        double d = numeric_limits<double>::max();
        for (int i = l; i < r; i++)
            for (int j = i+1; j <= r; j++) {
                double dij = rastojanje(tacke[i], tacke[j]);
                if (dij < d)
                    d = dij;
            }
        sort(next(begin(tacke), l), next(begin(tacke), r+1), porediY);
        return d;
    }

    int s = l + (r - l) / 2;
    double d2 = najblizeTacke(tacke, s+1, r);
    double d = min(d1, d2);
```

```
    merge(next(begin(tacke), l), next(begin(tacke), s+1),
          next(begin(tacke), s+1), next(begin(tacke), r+1),
          begin(pom), porediY);
    copy(begin(pom), end(pom), next(begin(tacke), l));

    double dl = tacke[s].x - d, dr = tacke[s].x + d;
    int k = 0;
    for (int i = l; i <= r; i++)
        if (dl <= tacke[i].x && tacke[i].x <= dr)
            pom[k++] = tacke[i];

    for (int j = i+1; j < pojas.size() && pojas[j].y - pojas[i].y <
    d; j++) {
        double dij = rastojanje(pojas[i], pojas[j]);
        if (dij < d)
            d = dij;
    }
    return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
    vector<Tacka> pom(tacke.size());
    return najblizeTacke(tacke, 0, tacke.size() - 1, pom);
}
```

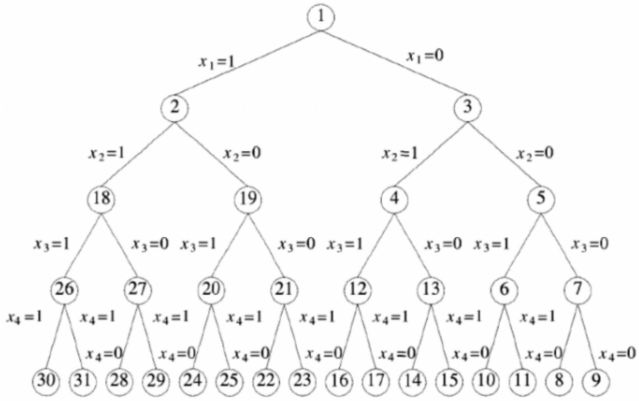
9 Pretraga

9.1 Generisanje kombinatornih objekata - rekurzivna pretraga (podskupovi, varijacije)

Problemi se često mogu rešiti iscrpnom pretragom, što podrazumeva da se ispitaju svi mogući kandidati za rešenja. Preduslov za to je da umemo sve te kandidate da nabrojimo. Iako u realnim primenama prostor potencijalnih rešenja može imati različitu strukturu, pokazuje se da je u velikom broju slučaja to prostor određenih klasičnih kombinatornih objekata: svih podskupova nekog konačnog skupa, svih varijacija (sa ili bez ponavljanja), svih kombinacija (sa ili bez ponavljanja), svih permutacija, svih particija i slično.
Objekti se obično predstavljaju n-torkama brojeva, pri čemu se isti objekti mogu torkama modelovati na različite načine. Na primer, svaki podskup skupa $\{a_1, ..., a_n\}$ se može predstaviti konačnim nizom indeksa elemenata koji mu pripadaju. Da bi svaki podskup bio jedinstveno predstavljen, potrebno je da taj niz bude kanonizovan (na primer, uređen rastući). Drugi način da se podskupovi predstave su n-torke logičkih vrednosti ili vrednost 0-1.
Svi objekti se obično mogu predstaviti drvetom i to drvo odgovara procesu njihovog generisanja tj. obilaska. Obilazak drveta se najjednostavnije izvodi u dubinu (često rekurzivno). Za prvu navedenu reprezentaciju podskupova drvo je dato na slici 1. Svaki čvor drveta odgovara jednom podskupu, pri čemu se odgovarajuća torka očitava na granama puta koji vodi od korena do tog čvora.



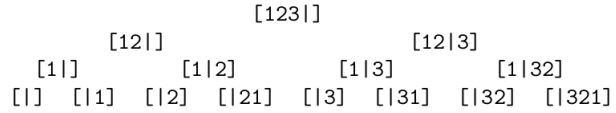
Za drugu navedenu reprezentaciju podskupova drvo je dato na slici 2. Samo listovi drveta odgovaraju podskupovima, pri čemu se odgovarajuća torka očitava na granama puta koji vodi od korena do tog čvora. Oba drveta sadrže 2^n čvorova kojima se predstavljaju podskupovi.
Prilikom generisanja objekata često je poželjno ređati ih određenim redom. S obzirom na to to da se svi kombinatorni objekti predstavljaju određenim torkama, prirodan poredak među njima je leksikografski poredak.



Problem: Napiši funkciju koja nabraja i obrađuje sve podskupove datog skupa čiji su elementi predstavljeni datim vektorom. Pretpostaviti da nam je na raspolaganju funkcija *obradi* koja obrađuje podskup (zadat takođe kao vektor).
Induktivno-rekurzivni pristup kaže da ako je skup prazan, onda je jedini njegov podskup prazan, a ako nije, onda se može razložiti na neki element i skup koji je za taj element manji od polaznog, čiji se podskupovi mogu odrediti rekurzivno. Svi podskupovi su onda ti koji su određeni za manji skup, kao i svi oni koji se od njih dobijaju dodavanjem tog izdvojenog elementa. Ovu konstrukciju nije jednostavno programski realizovati, jer se pretpostavlja da rezultat rada funkcije predstavlja skup svih podskupova skupa, a mi umesto funkcije želimo proceduru koja neće istovremeno čuvati sve podskupove već samo jedan po jedan nabrojati i obraditi. Do rešenja se može doći tako što se umesto da rekurzivni poziv vrati skup svih podskupova kojima se posle dodaje ili ne dodaje izdvojeni element, u jednom rekurzivnom pozivu prosledi parcijalno popunjeni podskup koji ne sadrži a u drugom rekurzivnom pozivu prosledi parcijalno popunjeni podskup koji sadrži izdvojeni element, a da svaki rekurzivni poziv ima zadatak da onda podskup koji je primio na sve moguće načine dopuni podskupovima elemenata smanjenog skupa.

```
void obradiSvePodskupove(const vector<int>& skup, const vector<int>&
    podskup) {
    if (skup.size() == 0) ispisi(podskup);
    else {
        int x = skup.back();
        vector<int> smanjenSkup = skup;
        smanjenSkup.pop_back();
        vector<int> podskupBez = podskup;
        obradiSvePodskupove(smanjenSkup, podskupBez);
        vector<int> podskupSa = podskup;
        podskupSa.push_back(x);
        obradiSvePodskupove(smanjenSkup, podskupSa);
    }
}
```

Ponašanje prethodne rekurzivne funkcije moguće je opisati drvetom. U svakom čvoru ćemo navesti uređen par (skup, podskup) razdvojene uspravnom crtom, koji odgovara tom rekurzivnom pozivu. U listovima drveta su skupovi prazni i obrađuju se dobijeni podskupovi.



Znatno bolja implementacija se može dobiti ako se isti niz tj. vektor koristi za predstavljanje skupova i podskupova, međutim, tada je potrebno obratiti pažnju na to da je nakon završetka svakog rekurzivnog poziva neophodno vratiti skup i podskup u početno stanje tj. u stanje koje je važilo na ulasku u funkciju. Dakle, invarijanta svakog rekurzivnog poziva biće to da ne menja ni skup ni podskup.

```
void obradiSvePodskupove(vector<int>& skup, vector<int>& podskup) {
    if (skup.size() == 0)
        obradi(podskup);
    else {
        int x = skup.back();
        skup.pop_back();
        obradiSvePodskupove(skup, podskup);
        podskup.push_back(x);
        obradiSvePodskupove(skup, podskup);
        podskup.pop_back();
        skup.push_back(x);
    }
}

void obradiSvePodskupove(vector<int>& skup) {
    vector<int> podskup;
    podskup.reserve(skup.size());
    obradiSvePodskupove(skup, podskup);
}
```

Bolje rešenje možemo dobiti tako što se izbacivanje elemenata iz skupa ne vrši tako što se gradi manji skup koji će se prosleđivati kroz rekurziju, nego se elementi skupa čuvaju u jedinstvenom vektoru koji se ne menja tokom rekurzije, a izbacivanje se postiže time što se održava brojač koji razdvaja elemente koji su izbačeni od onih koji su preostali u skupu. Sada nam je svejedno da li će izbačeni elementi biti na kraju ili na početku, pa biramo da brojač i razdvaja niz skup na dva dela: elementi na pozicijama [0, i) su izbačeni a na pozicijama [i, n) su preostali. I dužinu vektora u kome se čuva podskup ćemo u startu postaviti na n elemenata, pri čemu smatramo da se elementi nalaze na pozicijama iz intervala [0, j), dok su ostala mesta irelevantna. Pošto svaki rekurzivni poziv ima svoje kopije promenljivih i i j, nije potrebno efektivno restaurirati stanje skupa i podskupa na kraju poziva funkcije.

```
void obradiSvePodskupove(const vector<int>& skup, int i,
     vector<int>& podskup, int j) {
    if (i == skup.size())
        obradi(podskup, j);
    else {
        obradiSvePodskupove(skup, i + 1, podskup, j);
        podskup[j] = skup[i];
        obradiSvePodskupove(skup, i + 1, podskup, j + 1);
    }
}
```

Problem: Definisati funkciju koja obrađuje sve varijacije sa ponavljanjem dužine k skupa {1,..., n}.
Varijacije se mogu nabrojati induktivno rekurzivnom konstrukcijom. Jedina varijacija dužine nula je prazna. Sve varijacije dužine k dobijaju se od varijacija dužine *k-1* tako što se na poslednje mesto upišu svi mogući elementi od 1 do n. Ponovo ćemo implementaciju organizovati tako da što će umesto da vraća skup varijacija funkcija primati niz koji će na sve moguće načine dopunjavati varijacijama tekuće dužine n (koja će se smanjivati kroz rekurzivne pozive).

```
void obradiSveVarijacije(int k, int n, vector<int>& varijacija) {
    if (k == 0)
        obradi(varijacija);
    else
        for (int nn = 1; nn <= n; nn++) {
            varijacija[varijacija.size() - k] = nn;
            obradiSveVarijacije(k-1, n, varijacija);
        }
}
```

9.2 Generisanje kombinatornih objekata - rekurzivna pretraga (kombinacije sa i bez ponavljanja)

Problem: Definisati proceduru koja nabraja i obrađuje sve kombinacije bez ponavljanja dužine k skupa {1, 2,..., n}, za k $\leq$ n.
Da bi se izbeglo višestruko generisanje kombinacija koje su suštinski identične nametnućemo uslov da je svaka kombinacija predstavljena rastući sortiranim nizom brojeva. Zadatak rekurzivne funkcije biće da dopuni niz dužine k od pozicije i pa do kraja. Kada je i = k, niz je popunjen i potrebno je obraditi dobijenu kombinaciju. U suprotnom, biramo element koji ćemo postaviti na poziciju i. Pošto su kombinacije uređene strogo rastuće, on mora biti veći od prethodnog (ako prethodni ne postoji, onda može biti 1) i manji ili jednak n. Pošto su elementi strogo rastući, a od pozicije i pa do kraja niza treba postaviti *k-i* elemenata, najveći broj koji se može nalaziti na poziciji i je *n+i-k+1* i tada će na poziciji *k-1* biti vrednost n. U petlji stavljamo jedan po jedan od

tih elemenata na poziciju i i rekurzivno nastavljamo generisanje od naredne pozicije.

```
void obradiSveKombinacije(vector<int>& kombinacija, int i, int n) {
    int k = kombinacija.size();
    if (i == k) {
        obradi(kombinacija);
        return;
    }
    int pocetak = i == 0 ? 1 : kombinacija[i-1]+1;
    int kraj = n + i - k + 1;
    for (int x = pocetak; x <= kraj; x++) {
        kombinacija[i] = x;
        obradiSveKombinacije(kombinacija, i+1, n);
    }
}
```

Problem: Definisati proceduru koja nabraja i obrađuje sve kombinacije sa ponavljanjem dužine k skupa {1, 2,..., n}, za k $\leq$ n.
Rešenje se može dobiti krajnje jednostavnim prilagođavanjem rešenja kojim se generišu sve kombinacije bez ponavljanja. Kombinacije sa ponavljanjem se predstavljaju neopadajućim (umesto strogo rastućim) nizovima. Ponovo vršimo dva rekurzivna poziva. Jedan kada se na poziciju i stavi vrednost n_{min} i drugi kada se ta vrednost preskoči. Jedina razlika je što se prilikom postavljanja n_{min} na poziciji i ona ostavlja kao minimalna vrednost i u rekurzivnom pozivu koji se vrši, jer sada ne smeta da se ona ponovi - razlika je dakle samo u preskakanju uvećavanja jednog jedinog brojača.

```
void obradiSveKombinacijeSaPonavljanjem(vector<int>& kombinacija,
int i, int n_min, int n_max) {
    int k = kombinacija.size();
    if (i == k) {
        obradi(kombinacija);
        return;
    }
    if (n_min > n_max)
        return;
    kombinacija[i] = n_min;
    obradiSveKombinacijeSaPonavljanjem(kombinacija, i+1, n_min,
        n_max);
    obradiSveKombinacijeSaPonavljanjem(kombinacija, i, n_min+1,
        n_max);
}
```

9.3 Generisanje kombinatornih objekata - rekurzivna pretraga (permutacije)

Problem: Definisati proceduru koja nabraja i obrađuje sve permutacije skupa {1, 2,..., n}.
Rekurzivno generisanje permutacija u leksikografskom redosledu je veoma komplikovano, tako da ćemo se odreći uslova da permutacije moraju biti poređane leksikografski.Na prvu poziciju u nizu treba da postavljamo jedan po jedan element skupa, a zatim da rekurzivno određujemo sve permutacije preostalih elemenata. Fiksirane elemente i elemente koje treba permutovati možemo čuvati u istom nizu. Neka na pozicijama [0, k) čuvamo elemente koje treba permutovati, a na pozicijama [k, n) čuvamo fiksirane elemente. Razmatramo poziciju *k-1*. Ako je $k = 1$, tada postoji samo jedna permutacija jednočlanog niza na poziciji 0, nju pridružujemo fiskiranim elementima i ispisujemo permutaciju. Ako je $k > 1$, tada je situacija komplikovanija. Jedan po jedan element dela niza sa pozicija [0, k) treba da dovodimo na mesto *k-1* i da rekurzivno pozivamo permutovanje dela niza na pozicijama [0, k-1). Ideja koja se prirodno javlja je da vršimo razmenu elementa na poziciji *k-1* redom sa svim elementima iz intervala [0, k) i da nakon svake razmene vršimo rekurzivne pozive.Međutim, sa tim pristupom može biti problema. Naime, da bismo bili sigurni da će na poslednju poziciju stizati svi elementi niza, razmene moramo da vršimo u odnosu na početno stanje niza. Jedan način je da se pre svakog rekurzivnog poziva pravi kopija niza, ali postoji i efikasnije rešenje. Možemo kao invarijantu funkcije nametnuti da je nakon svakog rekurzivnog poziva raspored elemenata u nizu isti kao pre poziva funkcije. Ujedno to treba da bude i invarijanta petlje u kojoj se vrše razmene. Na ulasku u petlju raspored elemenata u nizu biće isti kao na ulasku u funkciju. Vršimo prvu razmenu, rekurzivno pozivamo funkciju i na osnovu invarijante rekurzivne funkcije znamo da će raspored nakon rekurzivnog poziva biti isti kao pre njega. Da bismo održali invarijantu petlje, potrebno je niz vratiti u početno stanje. Međutim, znamo da je niz promenjen samo jednom razmenom, tako da je dovoljno uraditi istu tu razmenu i niz će biti vraćen u početno stanje. Time je invarijanta petlje očuvana i može se preći na sledeću poziciju. Kada se petlja završi, na osnovu invarijante petlje znaćemo da je niz isti kao na ulazu u funkciju.

```
void obradiSvePermutacije(vector<int>& permutacija, int k) {
    if (k == 1)
        obradi(permutacija);
    else {
        for (int i = 0; i < k; i++) {
            swap(permutacija[i], permutacija[k-1]);
            obradiSvePermutacije(permutacija, k-1);
            swap(permutacija[i], permutacija[k-1]);
        }
    }
}
```

```
void obradiSvePermutacije(int n) {
    vector<int> permutacija(n);
    for (int i = 1; i <= n; i++)
        permutacija[i-1] = i;
    obradiSvePermutacije(permutacija, n);
}
```

9.4 Generisanje kombinatornih objekata - rekurzivna pretraga (particije)

Problem: Particija pozitivnog prirodnog broja n je predstavljanje broja n kao zbira nekoliko pozitivnih prirodnih brojeva pri čemu je redosled sabiraka nebitan. Napisati program koji ispisuje sve particije datog broja n sortirane leksikografski.
Svaka particija ima svoj prvi sabirak. Svakoj particiji broja n kojoj je prvi sabirak s ($1 \leq s \leq n$) 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. 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 generišemo sve particije broja n-s, već je potrebno da ojačamo induktivnu hipotezu. Pretpostavićemo da se u datom vektoru na pozicijama [0, i) nalaze ranije postavljeni elementi particije i da je zadatak procedure da taj niz dopuni na sve moguće načine particijama broja n u kojima su svi sabirci manji ili jednaki s_{max}. Izlaz iz rekurzije predstavljaće slučaj $n = 0$ u kom je jedina moguća particija broja 0 prazan skup, u kome nema sabiraka. Tada smatramo da je particija uspešno formirana i obrađujemo sadržaj vektora. Jedan način da se particije dopune je da se razmotre sve moguće varijante za sabirak na poziciji i. Na osnovu uslova oni moraju biti veći od nule, manji ili jednaki s_{max} i manji ili jednaki od n. Ako je m manji od brojeva n i s_{max}, mogući prvi sabirci su svi brojevi $1 \leq s' \leq m$. Kada fiksiramo sabirak s' niz rekurzivno dopunjavamo svim particijama broja n-s' u kojima su svi sabirci manji ili jednaki s', jer je potrebno preostali deo zbira predstaviti kao particiju brojeva koji nisu veći od s'.

```
void obradiParticije(int n, int smax, vector<int>& particija, int i)
    {
    if (n == 0)
        obradi(particija, i);
    else {
        for (int s = 1; s <= min(n, smax); s++) {
            particija[i] = s;
            obradiParticije(n-s, s, particija, i+1);
        }
    }
}
```

Razmotrimo još jedan, malo jednostavniji, način za rešavanje istog problema. Umesto da se analiziraju sve moguće vrednosti sabirka na poziciji i, moguće je razmatrati samo dve mogućnosti: prvu da se na poziciji i javlja sabirak s_{max} , a drugu da se na poziciji i javlja neki sabirak strogo manji od s_{max}. Prvi slučaj je moguć samo ako je n $\geq s_{max}$ i kada se na poziciju i postavi s_{max} niz dopunjujemo od pozicije *i+1* particijama broja *n*-s_{max} u kojima su svi sabirci manji ili jednaki s_{max}. Drugi slučaj je uvek moguć i tada particiju dopunjujemo particijama broja n u kojima je najveći sabirak s_{max}-1.

```
void obradiParticije(int n, int smax, vector<int>& particija, int i)
    {
    if (n == 0)
        obradi(particija, i);
    else {
        if (smax == 0)
            return;
        obradiParticije(n, smax-1, particija, i);
        if (n >= smax) {
            particija[i] = smax;
            obradiParticije(n-smax, smax, particija, i+1);
        }
    }
}
```

9.5 Generisanje kombinatornih objekata - naredni objekat u leksikografskom redosledu (podskupovi, varijacije)

Još jedan način da se nabroje svi kombinatorni objekti je da se definiše funkcija koja na osnovu datog objekta generiše sledeći (ili prethodni) u nekom redosledu. Definišimo tako funkciju koja na osnovu datog, određuje naredni podskup u leksikografskom redosledu. Postoje dva načina da se dođe do narednog podskupa. Jedan način je proširivanje kada se naredni podskup dobija dodavanjem nekog elementa u prethodni. Da bi dobijeni podskup sledio neposredno iza prethodnog u leksikografskom redosledu, dodati element podskupu mora biti najmanji mogući. Pošto je svaki podskup sortiran, element mora biti za jedan veći od poslednjeg elementa podskupa koji se proširuje. Jedini slučaj kada proširivanje nije moguće je kada je poslednji element podskupa najveći mogući. Drugi način je skraćivanje kada se naredni element dobija uklanjanjem nekih elemenata iz podskupa i izmenom preostalih elemenata. Iz podskupa se izbaci završni najveći element, a zatim se najveći od preostalih elemenata uveća za 1. Ako nakon izbacivanja najvećeg elementa ostane prazan skup, naredna kombinacija ne postoji.

```
bool sledeciPodskup(vector<int>& podskup, int n) {
    if (podskup.empty()) {
        podskup.push_back(1);
```

```
        return true;
    }
    if (podskup.back() < n) {
        podskup.push_back(podskup.back() + 1);
        return true;
    }
    podskup.pop_back();
    if (podskup.empty())  return false;
    podskup.back()++;
    return true;
}

void obradiPodskupove(int n, vector<int> podskup) {
    do {
        obradi(podskup);
    } while (sledeciPodskup(podskup, n));
}
```

Druga mogućnost je da se odredi naredna varijacija date varijacije u odnosu na leksikografski redosled. To se može učiniti tako što se uveća poslednji broj u varijaciji koji se može uvećati i da se nakon uvećavanja svi brojevi iza uvećanog broja postave na 1. Pozicija na kojoj se broj uvećava naziva se prelomna tačka. Jedan način implementacije je da prelomnu tačku nađemo linearnom pretragom od kraja niza, ako postoji da uvećamo element i da nakon toga do kraja niz popunimo jedinicama. Međutim, te dve faze možemo objediniti. Varijaciju obilazimo od kraja postavljajući na 1 svaki element u varijaciji koji je jednak broju n. Ako se zaustavimo pre nego što smo stigli do kraja niza, znači da smo pronašli element koji se može uvećati i uvećavamo ga. U suprotnom je varijacija imala sve elemente jednake n i bila je maksimalna u leksikografskom redosledu.

```
bool sledecaVarijacija(int n, vector<int>& varijacija) {
    int i;
    int k = varijacija.size();
    for (i = k-1; i >= 0 && varijacija[i] == n; i--)
        varijacija[i] = 1;
    if (i < 0)
        return false;
    varijacija[i]++;
    return true;
}

void obradiSveVarijacije(int k, int n) {
    vector<int> varijacija(k, 1);
    do {
        obradi(varijacija);
    } while(sledecaVarijacija(n, varijacija));
}
```

9.6 Generisanje kombinatornih objekata - naredni objekat u leksikografskom redosledu (kombinacije sa i bez ponavljanja)

Opišimo postupak kojim od date kombinacije možemo dobiti sledeću kombinaciju u leksikografskom redosledu. Ponovo tražimo prelomnu tačku tj. element koji se može uvećati. Pošto su kombinacije dužine k i organizovane su strogo rastuće, maksimalna vrednost na poslednjoj poziciji je n, na pretposlednjoj *n-1* itd. Dakle, poslednji element se može uvećati ako nije jednak n itd. Prelomna tačka je pozicija poslednjeg elementa koji je manji od svog maksimuma. Ako pozicije brojimo od 0, maksimum na poziciji *k-1* je n, na poziciji *k-2* je *n-1* itd. tako da je maksimum na poziciji i jednak *n-k+1+i*. Ako prelomna tačka ne postoji, naredna kombinacija u leksikografskom redosledu ne postoji. U suprotnom uvećavamo element na prelomnoj poziciji i da bismo nakon toga dobili leksikografski što manju kombinaciju, sve elemente iza njega postavljamo na najmanje moguće vrednosti. Pošto kombinacija mora biti sortirana strogo rastuće, nakon uvećanja prelomne vrednosti sve elemente iza nje postavljamo na vrednost koja je za jedan veća od vrednosti njoj prethodne vrednosti u nizu.

```
bool sledecaKombinacija(int n, vector<int>& kombinacija) {
    int k = kombinacija.size();
    int i;
    for (i = k-1; i >= 0 && kombinacija[i] == n; i--, n--);
    if (i < 0)
        return false;
    kombinacija[i]++;
    for (i++; i < k; i++)
        kombinacija[i] = kombinacija[i-1] + 1;
    return true;
}

void obradiSveKombinacije(int k, int n) {
    vector<int> kombinacija(k);
    for (int i = 0; i < k; i++)
        kombinacija[i] = i + 1;
    do {
        obradi(kombinacija);
    } while (sledecaKombinacija(n, kombinacija));
}
```

9.7 Generisanje kombinatornih objekata - naredni objekat u leksikografskom redosledu (permutacije)

U prvom koraku algoritma pronalazimo prvu poziciju i sdesna, takvu da je $a_i < a_{i+1}$ (za sve i+1 $\leq$ k < n-1 važi da je $a_k > a_{k+1}$). Ovo radimo najobičnijom linearnom pretragom. Ako takva pozicija ne postoji, naša permutacija je skroz opadajuća i samim tim leksikografski najveća. Nakon toga, je potrebno da

pronađemo najmanji element iza a_i koji je veći od a_i. Pošto je niz iza a_i opadajući, pronalazimo prvu poziciju j sdesna takvu da je $a_i < a_j$ i razmenjujemo elemente na pozicijama i i j. Pošto je ovom razmenom rep iza pozicije i i dalje striktno opadajući, da bismo dobili željenu leksikografski najmanju permutaciju potrebno je obrnuti redosled njegovih elemenata.

```
bool sledecaPermutacija(vector<int>& permutacija){
    int n = permutacija.size();
    int i = n - 2;
    while (i >= 0 && permutacija[i] > permutacija[i+1])
        i--;
    if (i < 0)
        return false;
    int j = n - 1;
    while (permutacija[j] < permutacija[i])
        j--;
    swap(permutacija[i], permutacija[j]);
    for (j = n - 1, i++; i < j; i++, j--)
        swap(permutacija[i], permutacija[j]);
    return true;
}
```

9.8 Iscrpna pretraga - definicija, svojstva

Gruba sila ili iscrpljujuća pretraga je opšta tehnika rešavanja problema koja se sastoji od sistematičnog nabrajanja svih mogućih kandidata za rešavanje i provere da li svaki kandidat zadovoljava problem. Dok je brut-force pretraga jednostavna za primenu, i uvek će pronaći rešenje ako postoji, njegova cena srazmerna je broju kandidata rešenja – što u mnogim praktičnim problemima pretenduje veoma brzim rastom kako se veličina problema povećava. Dakle, brute-force pretraga se obično koristi kada je veličina problema ograničena, ili kada postoji specifičan problem heuristike koji može biti iskorišćen da se smanji skup kandidata rešenja do veličine pogodne za rukovanje. Metoda se takođe koristi kada je jednostavnost implementacije važnija od brzine.

9.9 Iscrpna pretraga - n -dama

Jedan (naivan) način da se odrede svi mogući rasporedi je da se grubom silom nabroje svi mogući rasporedi, da se ispita koji od njih predstavljaju ispravan raspored (u kom se dame ne napadaju) i da se ispišu samo oni koji taj kriterijum zadovoljavaju. Najbolje rešenje se dobija ako se rasporedi predstave permutacijama elemenata skupa {1, 2,..., n}. Naime ako se dame ne napadaju, svaka od njih se nalazi bar u jednoj vrsti i bar u jednoj koloni. Ako i vrste i kolone obeležimo brojevima od 0 do *n-1*, tada je svakoj koloni jednoznačno pridružena vrsta u kojoj se nalazi dama u toj koloni i raspored možemo predstaviti nizom tih brojeva. Svakoj koloni je pridružena različita vrsta (jer dame ne smeju da

se napadaju), tako da je zaista u pitanju permutacija. Ovaj raspored u startu garantuje da se dame neće napadati ni horizontalno, ni vertikalno i jedino je potrebno odrediti da li se napadaju po dijagonali. Dve dame se nalaze na istoj dijagonali akko je horizontalni razmak između kolona u kojima se nalaze jednak vertikalnom razmaku vrsta. Za svaki par dama proveravamo da li se napadaju dijagonalno. Na taj način dobijamo narednu implementaciju.

```
bool dameSeNapadaju(const vector<int>& permutacija) {
    for (int i = 0; i < permutacija.size(); i++)
        for (int j = i + 1; j < permutacija.size(); j++)
            if (abs(i - j) == abs(permutacija[i] - permutacija[j]))
                return true;
    return false;
}

void obradi(const vector<int>& permutacija) {
    if (!dameSeNapadaju(permutacija)) {
        for (int x : permutacija)
            cout << x << " ";
        cout << endl;
    }
}
```

9.10 Iscrpna pretraga - tautologija istinitosnom tablicom

Gruba sila podrazumeva generisanje istinitosne tablice, izračunavanje vrednosti formule u svakoj vrsti (valuaciji) i proveri da li je formula u toj valuaciji tačna. Valuacije predstavljaju varijacije dužine n dvočlanog skupa tačno, netačno, gde je n ukupan broj promenljivih.
Pretpostavićemo da je formula predstavljena binarnim drvetom i da je dat pokazivač na koren, kao i funkcija izračunavanja vrednosti formule za datu valuaciju. U programu nećemo obraćati pažnju na učitavanje formule na osnovu tekstualne reprezentacije (parsiranje). Jednostavnosti radi istu strukturu ćemo koristiti i za skladištenje promenljivih i za skladištenje operatora. Strukturu i funkcije za kreiranje i oslobađanje čvorova je veoma jednostavno definisati. Ponovo koristimo pametne pokazivače, da ne bismo morali voditi računa o dealokaciji memorije.

```
enum TipCvora {PROM, I, ILI, NE, EKVIV, IMPL};

struct Cvor {
    TipCvora tip;
    int promenljiva;
    shared_ptr<Cvor> op1, op2;
};

typedef shared_ptr<Cvor> CvorPtr;
```

```
CvorPtr NapraviCvor() {
    return make_shared<Cvor>();
}

CvorPtr Prom(int p) {
    CvorPtr c = NapraviCvor();
    c->tip = PROM;
    c->promenljiva = p;
    return c;
}

CvorPtr Operator(TipCvora tip, CvorPtr op1, CvorPtr op2) {
    CvorPtr c = NapraviCvor();
    c->tip = tip;
    c->op1 = op1; c->op2 = op2;
    return c;
}

CvorPtr Ili(CvorPtr op1, CvorPtr op2) {
    return Operator(ILI, op1, op2);
}

CvorPtr I(CvorPtr op1, CvorPtr op2) {
    return cvorOperator(I, op1, op2);
}

bool vrednost(CvorPtr c, const vector<bool>& valuacija) {
    switch(c->tip) {
        case PROM: return valuacija[c->promenljiva];
        case I:    return vrednost(c->op1, valuacija) &&
                          vrednost(c->op2, valuacija);
        case ILI:  return vrednost(c->op1, valuacija) ||
                          vrednost(c->op2, valuacija);
        case EKVIV: return vrednost(c->op1, valuacija) ==
                          vrednost(c->op2, valuacija);
        case IMPL: return !vrednost(c->op1, valuacija) ||
                           vrednost(c->op2, valuacija);
        case NE:   return !vrednost(c->op1, valuacija);
    }
}

bool sledecaValuacija(vector<bool>& valuacija) {
    int i;
    for (i = valuacija.size() - 1; i >= 0 && valuacija[i]; i--)
        valuacija[i] = false;
    if (i < 0)   return false;
    valuacija[i] = true;
    return true;
}
```

```
int najvecaPromenljiva(CvorPtr formula) {
    if (formula->op1 == 0)
        return formula->promenljiva;
    int prom = najvecaPromenljiva(formula->op1);
    if (formula->op2 != nullptr) {
        int prom2 = najvecaPromenljiva(formula->op2);
        prom = max(prom, prom2);
    }
    return prom;
}

bool tautologija(CvorPtr formula) {
    vector<bool> valuacija(najvecaPromenljiva(formula) + 1, false);
    bool jeste = true;
    do {
        if (!vrednost(formula, valuacija))
            jeste = false;
    } while(jeste && sledecaValuacija(valuacija));
    return jeste;
}
```

9.11 Pretraga sa povratkom (bektreking) - definicija, svojstva

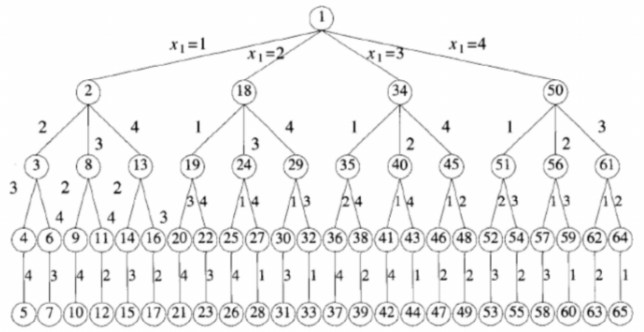
Algoritam pretrage sa povratkom (engl. *backtracking*) poboljšava tehniku grube sile tako što tokom implicitnog DFS obilaska drveta kojim se predstavlja prostor potencijalnih rešenja odseca one delove drveta za koje se unapred može utvrditi da ne sadrže ni jedno rešenje problema. Umesto da se čeka da se tokom pretrage stigne do lista i da se provera vrši tek tada, prilikom pretrage sa povratkom provera se vrši u svakom koraku i vrši se provera parcijalno popunjenih torki rešenja. Kvalitet rešenja zasnovanog na ovom obliku pretrage uveliko zavisi od kvaliteta funkcije kojom se vrši odsecanje. Ta funkcija mora biti potpuno precizna u svim čvorovima koji predstavljaju kandidate za rešenje i u tim čvorovima mora potpuno precizno odgovoriti da li je tekući kandidat zaista ispravno rešenje. Dodatno, ta funkcija procenjuje da li se trenutna torka može proširiti do ispravnog rešenja. U tom pogledu funkcija odsecanja ne mora biti potpuno precizna: moguće je da se odsecanje ne izvrši iako se torka ne može proširiti do ispravnog rešenja, međutim, ako se odsecanje izvrši, moramo biti apsolutno sigurni da se u odsečenom delu zaista ne nalazi ni jedno ispravno rešenje. Ako je funkcija odsecanja takva da odsecanje ne vrši nikada, bektreking algoritam se svodi na algoritam grube sile.
Formulišimo opštu shemu rekurzivne implentacije pretrage sa povratkom. Pretpostavljamo da su parametri procedure pretrage trenutna torka rešenja i njena duzina, pri čemu je niz alociran tako da se u njega može smestiti i najduže rešenje. Takođe, pretpostavljamo da na raspolaganju imamo funkciju osecanje

koja proverava da li je trenutna torka kandidat da bude rešenje ili deo nekog rešenja. Pretpostavljamo i da znamo da li trenutna torka predstavlja rešenje. Na kraju, pretpostavljamo i da za svaku torku dužine k možemo eksplicitno odrediti sve kandidate za vrednost na poziciji k. Rekurzivnu pretragu tada možemo realizovati narednim (pseudo)kodom.

```
void pretraga(const vector<int>& v, int k) {
    if (odsecanje(v, k))
        return;
    if (jestePotencijalnoResenje(v, k))
        ispisi(v, k);
    if (k == v.size())
        return;
    for (int x : kandidati(v, k)) {
        v[k] = x;
        pretraga(v, k+1);
    }
}
```

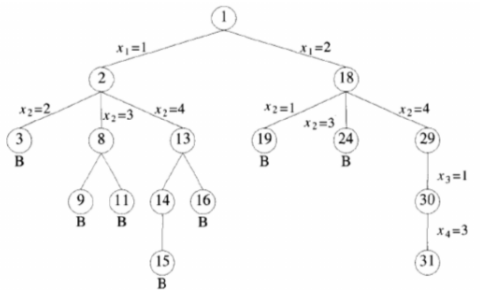
9.12 Pretraga sa povratkom - n-dama

Ceo prostor potencijalnih rešenja se može predstaviti drvetom u čijim se listovima nalaze permutacije koje predstavljaju rasporedi dama. Rešenje grubom silom ispituje sve listove ovog drveta.



Osnovna razlika ovog u odnosu na prethodno rešenje biće to što se korektnost permutacija neće proveravati tek nakon što su generisane u celosti, već će biti vršena korektnost i svake delimično popunjene permutacije. U mnogim slučajevima veoma rano će biti otkriveno da su postavljene dame na istoj dijagonali i cela ta grana pretrage biće napuštena, što daje potencijalno velike dobitke u efikasnosti. Drvo prostora pretrage ćemo obilaziti u dubinu, odsecajući sve one delove za koje utvrdimo da ne mogu dovesti do uspešnog rešenja. Deo prostora pretrage sasečenog na ovaj način prikazano je na narednoj slici. Pretraga počinje sa postavljanjem dame u prvoj vrsti u prvu kolonu (ovo ćemo predstaviti četvorkom (1???)). Nakon toga postavljamo damu u drugoj vrsti

u drugu kolonu (12??), međutim, odmah detektujemo da se dame napadaju i odsecamo tu granu pretrage. Pokušavamo sa postavljanjem dame u drugoj vrsti u treću kolonu (13??). Nastavljamo tako što damu u trećoj koloni postavljamo u drugu vrstu (132?), međutim ona se napada damom u drugoj vrsti i tu granu odsecamo. Pokušavamo i sa postavljanjem dame u trećoj vrsti u četvrtu kolonu (134?), međutim i tada se ona napada sa damom u drugoj vrsti, pa i ovu granu odsecamo.



Osnovna rekurzivna funkcija primaće vektor u kome se na pozicijama iz intervala [0, k) nalaze dame koje su postavljene u prvih k kolona i zadatak funkcije će biti da ispiše sve moguće rasporede koji proširuju taj. Važna invarijanta će biti da se dame koje su postavljene u tih prvih k kolona ne napadaju. Ako je $k = n$, tada su sve dame već postavljene, na osnovu invarijante znamo da se ne napadaju i možemo da ispišemo to rešenje. U suprotnom, razmatramo sve opcije za postavljanje dame na poziciju k, tako da se dame u tako proširenom skupu ne napadaju. Pošto se zna da se dame na pozicijama [0, k) ne napadaju potrebno je samo proveriti da li se dama na poziciji k napada sa nekom od dama postavljenih u prvih k kolona. Razmotrimo šta su kandidati za vrednosti na poziciji k. Pošto ne smemo imati dva ista elementa niza tj. dve iste vrste na kojima se nalaze dame, mogli bismo u delu niza na pozicijama [k, n) održavati skup elemenata koji su potencijalni kandidati za poziciju k. Međutim, pošto za proveru dijagonala moramo uporediti damu k sa svim prethodno postavljenim damama, implementaciju možemo olakšati tako što na poziciju k stavljamo širi skup mogućih kandidata (skup svih vrsta od 0 do n-1), a onda za svaki od tih brojeva proveravamo da li se javio u prethodnom delu niza čime bi se dame napadale po horizontali i da li se postavljanjem dame na u tu vrstu ona po dijagonali napadala sa nekom prethodno postavljenom damom. Ako se ustanovi da to nije slučaj, onda je invarijanta zadovoljena i rekurzivno prelazimo na popunjavanje narednih dama. Nema potrebe za eksplicitnim poništavanjem odluka koje donesemo, jer će se u svakoj novoj iteraciji dopuštena vrednost upisati na poziciju k, automatski poništavajući vrednost koju smo tu ranije upisali.

```
bool dameSeNapadaju(const vector<int>& permutacija, int k) {
    for (int i = 0; i < k; i++) {
        if (permutacija[i] == permutacija[k])
            return true;
```

```
        if (abs(k-i) == abs(permutacija[k] - permutacija[i]))
            return true;
    }
    return false;
}

void nDama(vector<int>& permutacija, int k) {
    if (k == permutacija.size())
        ispisi(permutacija);
    else {
        for (int i = 0; i < permutacija.size(); i++) {
            permutacija[k] = i;
            if (!dameSeNapadaju(permutacija, k))
                nDama(permutacija, k+1);
        }
    }
}
```

9.13 Pretraga sa povratkom - sudoku

Problem: Sudoku je zagonetka u kojoj se zahteva da se brojevi od 1 do 9 rasporede po polju dimenzije 9x9 tako da se u svakoj vrsti, svakoj koloni i svakom od 9 kvadrata dimenzije 3x3 nalaze različiti brojevi. Napisati funkciju koja na osnovu nekoliko datih početnih vrednosti određuje da li je moguće ispravno popuniti Sudoku i ako jeste, određuje bar jedno rešenje.
Sudoku se može uopštiti na polje dimenzije n^2xn^2 . Osnovni problem se dobija za n = 3. I ovaj se problem može rešiti klasičnom pretragom sa povratkom. Polja ćemo popunjavati određenim redosledom. Najprirodnije je da to bude vrstu po vrstu, kolonu po kolonu. Jednostavno možemo definisati funkciju koja određuje koje polje je potrebno popuniti posle polja sa koordinatama (i, j) – uvećava se j i ako dostigne vrednosti n^2 , vraća se na nulu, a uvećava se i.
Rekurzivnu funkciju pretrage definišemo tako da pokušava da dopuni matricu podrazumevajući da su sva polja pre polja (i, j) u redosledu koji smo opisali već popunjena tako da među trenutno popunjenim poljima nema konflikata. Vrednosti 0 u matrici označavaju prazna polja. Ako je polje (i, j) već popunjeno, proverava se da li je to poslednje polje i ako jeste, funkcija vraća da je cela matrica uspešno popunjena. Ako to nije poslednje polje, samo se prelazi na popunjavanje narednog polja. Ako je polje prazno, tada pokušavamo da na njega upišemo sve vrednosti od 1 do n^2. Nakon upisa svake od vrednosti proveravamo da li je na taj način napravljen konflikt. Pošto na osnovu invarijante znamo da konflikt nije postojao ranije, dovoljno je samo proveriti da li novo upisana vrednost pravi konflikt tj. da li je ista ta vrednost već upisana u vrsti i, koloni j ili kvadratu u kom se nalazi polje (i, j).

```
const int n = 3;
```

```
bool konflikt(const vector<vector<int>>& m, int i, int j) {
    for (int k = 0; k < n * n; k++)
        if (k != i && m[i][j] == m[k][j])
            return true;
    for (int k = 0; k < n * n; k++)
        if (k != j && m[i][j] == m[i][k])
            return true;
    int x = i / n, y = j / n;
    for (int k = x * n; k < (x + 1) * n; k++)
        for (int l = y * n; l < (y + 1) * n; l++)
            if (k != i && l != j && m[i][j] == m[k][l])
                return true;
    return false;
}

void sledeci(int& i, int& j) {
    j++;
    if (j == n * n) {
        j = 0;
        i++;
    }
}

bool sudoku(vector<vector<int>>& m, int i, int j) {
    if (m[i][j] != 0) {
        if (i == n * n - 1 && j == n * n - 1)
            return true;
        sledeci(i, j);
        return sudoku(m, i, j);
    }
    else {
        for (int k = 1; k <= n*n; k++) {
            m[i][j] = k;
            if (!konflikt(m, i, j))
                if (sudoku(m, i, j))
                    return true;
        }
        m[i][j] = 0;
        return false;
    }
}
```

9.14 Pretraga sa povratkom - izlazak iz lavirinta (BFS, DFS)

Problem: Napisati program koji određuje da li je u lavirintu moguće stići od starta do cilja. Lavirint je određen matricom karaktera (x označava zid kroz koji se ne može proći i matrica je ograničena sa četiri spoljna zida, . označava slobodna polja, S označava start, a C označava cilj). Sa svakog polja dozvoljeno je kretanje u četiri smera (gore, dole, levo i desno).
Zadatak rešavamo iscrpnom pretragom svih mogućih putanja. Pretragu možemo organizovati "u dubinu". Funkcija prima startno i ciljno polje, pri čemu se startno polje menja tokom rekurzije. Ako je startno polje poklapa sa ciljnim, put je uspešno pronađen. U suprotnom, ispitujemo 4 suseda startnog polja i ako je susedno polje slobodno (nije zid), pretragu rekurzivno nastavljamo od njega (sudedno polje postaje novo startno polje). Potrebno je da obezbedimo da se već posećena startna polja ne obrađuju ponovo i za to koristimo pomoćnu matricu u kojoj za svako polje registrujemo da li je posećeno ili nije. Pre rekurzivnog proveravamo da li je susedno polje posećeno i ako jeste, rekurzivni poziv preskačemo, a ako jeste označavamo da je to polje posećeno.

```
bool pronadjiPut(Matrica<bool>& lavirint, Matrica<bool>& posecen,
int x1, int y1, int x2, int y2) {
    if (x1 == x2 && y1 == y2)
        return true;
    int pomeraj[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    for (int i = 0; i < 4; i++) {
        int x = x1 + pomeraj[i][0], y = y1 + pravac[i][1];
        if (!lavirint[x][y] && !posecen[x][y]) {
            posecen[x][y] = true;
            if (pronadjiPut(lavirint, posecen, x, y, x2, y2))
                return true;
        }
    }
    return false;
}

bool pronadjiPut(Matrica<bool>& lavirint, int x1, int y1, int x2,
int y2) {
    int m = lavirint.size(), n = lavirint[0].size();
    Matrica<bool> posecen = napraviMatricu(m, n, false);
    posecen[x1][y1] = true;
    return pronadjiPut(lavirint, posecen, x1, y1, x2, y2);
}
```

Druga mogućnost je da se pretraga vrši "u širinu" što znači da sva polja obrađujemo u rastućem redosledu rastojanja od početnog startnog polja. Prvo je potrebno obraditi sva susedna polja startnog polja, zatim sva njihova susedna polja i tako dalje. Implementaciju vršimo pomoću reda u koji postavljamo polja koja treba obraditi. Na početku u red postavljamo samo ciljno polje. Skidamo

jedan po jedan element iz reda, proveravamo da li je to ciljno polje i ako jeste, prekidamo funkciju vraćajući rezultat. Ponovo moramo voditi računa o tome da ista polja ne posećujemo više puta. Ako dodatno želimo da odredimo najkraće rastojanje od startnog do ciljnog polja, možemo čuvati pomoćnu matricu u kojoj za svako polje registrujemo to rastojanje. Za polja koja su još neposećena možemo upisati neku negativnu vrednost. Kada tekući element skinemo iz reda, njegovo rastojanje od početnog polja možemo očitati iz te matrice, a zatim, za sve njegove ranije neposećene susede možemo u tu matricu upisati da im je najkraće rastojanje za jedan veće od najkraćeg rastojanja tekućeg čvora.

```
int najkraciPut(Matrica<bool>& lavirint, int x1, int y1, int x2, int
    y2) {
    int m = lavirint.size(), n = lavirint[0].size();
    Matrica<int> rastojanje = napraviMatricu(m, n, -1);
    rastojanje[x1][y1] = 0;
    queue<pair<int, int>> red;
    red.push(make_pair(x1, y1));
    while (!red.empty()) {
        x1 = red.front().first;
        y1 = red.front().second;
        red.pop();
        int r = rastojanje[x1][y1];
        if (x1 == x2 && y1 == y2)
            return r;
        int pomeraj[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
        for (int i = 0; i < 4; i++) {
            int x = x1 + pomeraj[i][0], y = y1 + pomeraj[i][1];
            if (!lavirint[x][y] && rastojanje[x][y] == -1) {
                rastojanje[x][y] = r + 1;
                red.push(make_pair(x, y));
            }
        }
    }
    return -1;
}
```

9.15 Pretraga sa povratkom - podskup elemenata datog zbira (varijanta 0-1 ranca)

Problem: Dat je skup pozitivnih brojeva. Napisati program koji određuje koliko elemenata tog skupa ima zbir jednak datom pozitivnom broju.
Zadatak jednostavno možemo rešiti iscrpnom pretragom, tako što generišemo sve podskupove i prebrojimo one čiji je zbir elemenata jednak traženom. To možemo uraditi bilo rekurzivnom funkcijom koja nabraja sve podskupe, bilo nabrajanjem svih podskupova zasnovanom na pronalaženju narednog podskupa u leksikografskom redosledu. U prvom slučaju kao parametar rekurzivne funkcije možemo prosleđivati trenutni ciljni zbir tj. razliku između traženog zbira i zbira

elemenata trenutno uključenih u podskup. Sam podskup nije neophodno održavati. Ako je ciljni zbir jednak nuli, to znači da je zbir trenutnog podskupa jednak traženom i da smo našli jedan zadovoljavajući podskup. U suprotnom, ako u skupu nema preostalih elemenata, tada znamo da nije moguće napraviti podskup traženog zbira. U suprotnom uklanjamo trenutni element iz skupa i razmatramo mogućnost da se on uključi u podskup i da se ne uključi. U prvom slučaju umanjujemo ciljni zbir za vrednost tog elementa, a u drugom ciljni zbir ostaje nepromenjen. U oba slučaja je skup smanjen.

```
int brojPodskupovaDatogZbira(const vector<int>& skup,
int ciljniZbir, int k) {
    if (ciljniZbir == 0)
        return 1;
    if (k == skup.size())
        return 0;
    return brojPodskupovaDatogZbira(skup, ciljniZbir - skup[k], k+1)
         + brojPodskupovaDatogZbira(skup, ciljniZbir, k+1);
}
```

Efikasnije rešenje se može dobiti ako se primeni nekoliko različitih tipova odsecanja u pretrazi. Ključna stvar je da odredimo interval u kome mogu ležati zbirovi svih podskupova trenutnog skupa. Pošto su svi elementi pozitivni, najmanja moguća vrednost zbira podskupa je nula (u slučaju praznog skupa), dok je najveća moguća vrednost zbira podskupa jednaka zbiru svih elemenata skupa. Dakle, ako je ciljni zbir strogo manji od nule ili strogo veći od zbira svih elemenata trenutnog skupa, tada ne postoji ni jedan podskup čiji je zbir jednak ciljnom. Umesto da zbir svih elemenata niza računamo iznova u svakom rekurzivnom pozivu, možemo primetiti da se u svakom narednom rekurzivnom pozivu skup samo može smanjiti za jedan element, pa se zbir može računati inkrementalno, umanjivanjem zbira polaznog skupa za jedan izbačeni element. Ako pretpostavimo da je niz sortiran, tada je najmanji element uvek prvi u preostalom delu niza. Ovo nam omogućava da dodamo još jedno odsecanje, koje se može ispostaviti kao značajno.

```
int brojPodskupovaDatogZbira(const vector<int>& skup, int ciljniZbir,
int zbirPreostalih, int k) {
    if (ciljniZbir == 0)
        return 1;
    if (zbirPreostalih < ciljniZbir)
        return 0;
    if (skup[k] > ciljniZbir)
        return 0;
    return brojPodskupovaDatogZbira(skup, ciljniZbir - skup[k],
                                    zbirPreostalih - skup[k], k+1)
         + brojPodskupovaDatogZbira(skup, ciljniZbir,
                                    zbirPreostalih - skup[k], k+1);
}
```

```
int brojPodskupovaDatogZbira(vector<int>& skup, int ciljniZbir) {
    int n = skup.size();
    sort(begin(skup), end(skup));
    int zbirSkupa = accumulate(begin(skup), end(skup), 0);
    return brojPodskupovaDatogZbira(skup, ciljniZbir, zbirSkupa, 0);
}
```

9.16 Pretraga sa povratkom - merenje sa n tegova

Problem: Dato je n tegova i za svaki teg je poznata njegova masa. Datim tegovima treba što preciznije izmeriti masu S. Napisati program koji određuje najmanja razlika pri takvom merenju.
Osnovu čini rekurzivna provera svih podskupova. Najbolji rezultat dobijen u jednoj grani pretrage upotrebićemo za odsecanje druge grane pretrage. Naime, vrednost minimalne razlike dobijene u jednom rekurzivnom pozivu možemo upotrebiti za eventualno odsecanje drugog rekurzivnog poziva. Ako tekuća masa uvećana za trenutni teg prevazilazi ciljnu masu za iznos veći od minimalne razlike, onda će to biti slučaj i sa svim proširenjima tog podskupa tegova, tako da nema potrebe za obradom tog skupa i rekurzivni poziv se može preskočiti. Takođe, ako je tekuća masa uvećana za masu svih preostalih tegova manja od ciljne mase više od iznosa minimalne razlike, ponovo je moguće izvršiti odsecanje.

```
double merenje(const vector<double>& tegovi, double ciljnaMasa,
int k, double tekucaMasa, double preostalaMasa) {
    if (k == tegovi.size())
        return abs(ciljnaMasa - tekucaMasa);
    double minRazlika = merenje(tegovi, ciljnaMasa, k+1,
                  tekucaMasa, preostalaMasa - tegovi[k]);
    if (tekucaMasa + preostalaMasa > ciljnaMasa - minRazlika &&
        tekucaMasa + tegovi[k] < ciljnaMasa + minRazlika) {
           double razlika = merenje(tegovi, ciljnaMasa, k+1,
             tekucaMasa + tegovi[k], preostalaMasa - tegovi[k]));
           if (razlika < minRazlika)
             minRazlika = razlika;
    }
    return minRazlika;
}

double merenje(const vector<double>& tegovi, double ciljnaMasa) {
    double ukupnaMasa = 0.0;
    for (int i = 0; i < tegovi.size(); i++)
        ukupnaMasa += tegovi[i];
    return merenje(tegovi, ciljnaMasa, 0, 0.0, ukupnaMasa);
}
```

9.17 Pretraga sa povratkom - 3-bojenje grafa

Problem: Za dati graf zadat listom svojih grana (parova čvorova) napisati program koji proverava da li se taj graf može obojiti sa tri različite boje, tako da su svi susedni čvorovi obojeni različitim bojama.

```
bool oboj(const vector<vector<int>>& susedi, int cvor,
vector<int>& boje) {
    int brojCvorova = susedi.size();
    if (cvor >= brojCvorova)
        return true;
    for (int boja = 1; boja <= 3; boja++) {
        bool mozeBoja = true;
        for (int sused : susedi[cvor])
            if (boje[sused] == boja)
                mozeBoja = false;
        if (mozeBoja) {
            boje[cvor] = boja;
            if (oboj(susedi, cvor+1, boje))
                return true;
        }
    }
    return false;
}

bool oboj(const vector<vector<int>>& susedi, vector<int>& boje) {
    int brojCvorova = susedi.size();
    int cvor = 0;
    while (cvor < susedi.size() && susedi[cvor].size() == 0)
        boje[cvor] = 1;
    boje[cvor] = 1; boje[susedi[0][0]] = 2;
    return oboj(susedi, cvor, boje);
}
```

10 Dinamičko programiranje

10.1 Dinamičko programiranje - definicija, oblici, svojstva, primeri

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. 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 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.

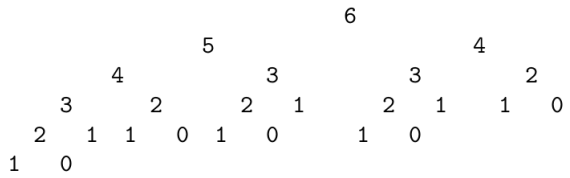
10.2 Dinamičko programiranje - Fibonačijevi brojevi

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 $F_n = Fn - 1 + F + n - 2$, za n>1.

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

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:



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.

```
int fib(int n, map<int, int>& memo) {
    auto it = memo.find(n);
    if (it != memo.end())
        return it->second;
    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 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, niz može biti čak i memorijski efikasniji u odnosu na mapu.
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 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) {
    if (memo[n] != -1)
        return memo[n];
    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) {
    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:

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. I u ovom slučaju je složenost izračunavanja $O(n)$.

```
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];
}
```

Pažljivijom analizom možemo ustanoviti da nam niz zapravo i nije potreban. 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++) {
        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:

1. Induktivno-rekurzivnom konstrukcijom konstruiše se rekurzivna definicija koja je neefikasna jer se isti pozivi prekapaju.
2. Tehnikom memoizacije poboljšava se složenost tako što se u pomoćnom rečniku 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 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 više nisu potrebne, tako da se umesto istovremnog pamćenja svih elemenata pamti samo nekoliko prethodnih.

10.3 Dinamičko programiranje - prebrojavanje kombinatornih objekata

Broj kombinacija

Problem: Napiši program koji određuje broj kombinacija dužine k iz skupa od n elemenata.
Možemo krenuti od funkcije za generisanje svih 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) {
    if (i == k)
        return 1;
    if (k - i > n_max - n_min + 1)
        return 0;
    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. Ako te dve veličine zamenimo sa k tj. n dobijamo narednu definiciju:

```
int brojKombinacija(int k, int n) {
    if (k == 0) return 1;
    if (k > n) return 0;
    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. Međutim, u slučaju poziva funkcije za k=n dobiće se uvek povratna vrednost 1, 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.

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

Primećujemo da smo ovom transformacijom dobili čuvene osobine binomnih koeficijenata. 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.
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 $\begin{bmatrix} n \\ k \end{bmatrix}$ 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. Pošto nas neće zanimati vrednosti veće od polaznog k i pošto se i k i n smanjuju tokom rekurzije, možemo odseći deo trougla desno od k.

```
int brojKombinacija(int k, int n, vector<vector<int>>& memo) {
    if (memo[n][k] != 0) return memo[n][k];
    if (k == 0 || k == n) return memo[n][k] = 1;
    return memo[n][k] = brojKombinacija(k-1, n-1, memo) +
                        brojKombinacija(k, n-1, memo);
}

int brojKombinacija(int K, int N) {
    vector<vector<int>> memo(N+1);
    for (int n = 0; n <= N; n++)
        memo[n].resize(min(K+1, n+1), 0);
    return brojKombinacija(K, N, memo);
}
```

Umesto memoizacije možemo upotrebiti i dinamičko programiranje naviše, oslobiditi se rekurzije i popuniti trougao vrstu po vrstu naniže.

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

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

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

Pažljivijom analizom prethodnog koda vidimo da 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). 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 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. Memorijska složenost ovog rešenja je *O(k)*, dok je vremenska *O(nk)*.

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

Broj kombinacija sa ponavljanjem

Opet se jednostavnim transformacijama koda koji vrši njihovo generisanje može doći do naredne rekurzivne definicije. 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. 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];
}
```

Broj particija

Problem: Particija pozitivnog prirodnog broja n je predstavljanje broja n na zbir nekoliko pozitivnih prirodnih brojeva pri čemu je redosled sabiraka nebitan. Napisati program koji određuje broj particija za dati prirodan broj n.
Algoritmi za određivanje broja particija odgovaraju algoritmima za generisanju svih particija.

```
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) x (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);
}
```

Umesto memoizacije možemo upotrebiti i dinamičko programiranje naviše. 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.

```
int brojParticija(int N) {
    vector<vector<int>> dp(N+1);
    for (int n = 0; n <= N; n++)
        dp[n].resize(N+1);
    for (int smax = 0; smax <= N; smax++)
        dp[0][smax] = 1;
    for (int n = 1; n <= N; n++)
        dp[n][0] = 0;
    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. Međutim, ako matricu popunjavamo kolonu po kolonu odozgo naniže, možemo dobiti memorijsku složenost $O(n)$. Svaki element zavisi od elementa u istoj vrsti u prethodnoj koloni i elementa u istoj koloni u nekoj od prethodnih vrsta, tako da ako možemo čuvati samo dve uzastopne kolone. 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.

```
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];
}
```

10.4 Dinamičko programiranje - podskup elemenata datog zbira (varijanta 0-1 ranca)

Problem: Dato je n predmeta čije su mase $m_0, ..., m_{n-1}$ i ranac nosivosti M. 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 $O(2^n)$. 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) {
    if (M == 0) return true;
    if (n == 0) return false;
    if (zbirPodskupa(m, n-1, M)) return true;
    if (m[n-1] <= M && zbirPodskupa(m, n-1, M - m[n-1]))
        return true;
    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. 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).
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)* x *(M + 1)*. 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 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];
}
```

Pretragu možemo zaustaviti čim vrednost u koloni M prvi put postane tačna. 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.

```
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. 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];
}
```

Memorijska složenost je *O(M)*, dok je vremenska složenost *O(nM)*. U pitanju je algoritam čija vremenska (a i memorijska) složenost eksponencijalno zavisi od veličine ulaza. Za relativno male vrednosti M, ovaj algoritam se ponaša efikasno. Ovakvi algoritmi se nazivaju pseudo-polinomijalni. Složenost postupka rekonstrukcije je *O(M + n)*.

10.5 Dinamičko programiranje - kusur sa minimalnim brojem novčića

Problem: Na raspolaganju imamo n novčića čiji su iznosi celi brojevi $m_0, ..., m_{n-1}$. 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 i pretragom sa odsecanjima. 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. 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. 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}.
Krenimo od memoizovane verzije pretrage sa odsecanjem. Za memoizaciju možemo upotrebiti matricu.

```
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);
}
```

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.

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

10.6 Dinamičko programiranje - maksimalni zbir segmenta (veza sa Kadanovim algoritmom)

Problem: Definisati efikasnu funkciju koja pronalazi najveći mogući zbir segmenta (podniza uzastopnih elemenata) datog niza brojeva.
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 $0 < \text{i} \leq \text{n}$. 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.

```
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();
    vector<int> dp(n + 1);
    dp[0] = 0;
    for (int i = 1; i <= n; i++)
        p[i] = max(0, dp[i-1] + a[i-1]);
    int rez = dp[0];
    for (int i = 1; i <= n; i++)
        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.

10.7 Dinamičko programiranje - najduži zajednički podniz, najduža zajednička podniska

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. 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. U ovom problemu zadovoljeno je svojstvo optimalne podstrukture.

Pošto rekurzija teče po prefiksima niski, jedini promenljivi parametri tokom rekurzije mogu biti dužine tih prefiksa.

```
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.

```
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];
}
```

Vremenska složenost ovog algoritma jednaka je $O(n_1 \cdot n_2)$, a memorijska složenost je $O(n_2)$.

Najduži 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.
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. 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. 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;
}
```

Naravno, i ovde možemo upotrebiti memorijsku optimizaciju. Pošto element više ne zavisi od prethodnog elementa u tekućoj vrsti, elemente možemo ažurirati sdesna na levo. Vremenska složenost ovog algoritma je $O(n_1 \cdot n_2)$, dok je memorijska složenost $O(n_2)$.

10.8 Dinamičko programiranje - 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});
    }
}
```

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 $(n_1 + 1)$ x $(n_2 + 1)$, a konačan rezultat će se nalaziti na mestu (n_1, n_2). 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];
}
```

10.9 Dinamičko programiranje - 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 razmatramo 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 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.

```
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));
}
```

U prethodnoj funkciji dolazi do preklapanja rekurzivnih poziva, pa je poželjno upotrebiti memoizaciju. Za memoizaciju koristimo matricu.

```
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));
}
```

Do efikasnog rešenja možemo doći i dinamičkim programiranjem odozdo naviše. Ovo rešenje ima i memorijsku i vremensku složenost $O(n^2)$.

```
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];
}
```

10.10 Dinamičko programiranje - najduži rastući podniz

Problem: Napisati program koji određuje dužinu najdužeg strogo rastućeg podniza (ne obavezno uzastopnih elemenata) u datom nizu celih brojeva.
Zadatak rešavamo induktivno-rekurzivnom konstrukcijom. 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 $O(n)$. 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 $1 \leq \mathrm{d} \leq d_{max}$ odredimo najmanji element kojim se završava rastući podniz dužine d. Primetimo da niz tih vrednosti uvek strogo rastući.

```
int najduziRastuciPodniz(const vector<int>& a) {
    int n = a.size();
    vector<int> dp(n);
    int max = 0;
    for (int i = 0; i < n; i++) {
        auto it = lower_bound(dp.begin(), next(dp.begin(), max),
            a[i]);
        int d = distance(dp.begin(), it);
        dp[d] = a[i];
        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 $O(nlogn)$. Ova granica je asimptoski precizna, jer taj najgori slučaj zapravo nastupa u slučaju strogo rastućih nizova.

10.11 Dinamičko programiranje - optimalni raspored zagrada

Problem: Množenje matrica dimenzije D_1 x D_2 i dimenzije D_2 x D_3 daje matricu dimenzije D_1 x 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_{n-1} određuje minimalni broj množenja brojeva prilikom množenja matrica dimenzija D_0 x D_1, D_1 x D_2,..., D_{n-2} x D_{n-1}.
Krenimo od rekurzivnog rešenja. Kako god da grupišemo matrice neko množenje je to koje se poslednje izvršava. Osnovna ideja je da ekplicitno izanaliziramo sve 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.

```
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;
}
```

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 x 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;
}
```

10.12 Dinamičko programiranje - 0-1 problem ranca

Problem: Dato je n predmeta čije su mase celi brojevi m_0,..., m_{n-1} i cene realni brojevi c_0,..., c_{n-1}. Napisati program koji određuje najveću cenu koja se može pokupiti pomoću ranca celobrojne nosivosti M.
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 dolazi u obliku dinamičkog programiranja. 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 $O(NM)$, gde je N broj predmeta. Za ovakve algoritme se kaže da su pseudo-polinomijalni. Veličina ulaza vezanog za broj M odgovara broju cifara broja M, a vreme izvršavanja algoritma eksponencijalno raste u odnosu na taj broj.

11 Pohlepni algoritmi

11.1 Pohlepni algoritmi - definicija, svojstva, primeri

Algoritmi kod kojih se u svakom koraku uzima lokalno optimalno rešenje i koji garantuju da će takvi izbori na kraju dovesti do globalno optimalnog rešenja nazivaju se *pohlepni* ili *gramzivi algoritmi*. Pohlepni algoritmi ne vrše ispitivanje različitih slučajeva niti iscrpnu pretragu i stoga su po pravilu veoma efikasni. Takođe, obično se veoma jednostavno implementiraju. Sa druge strane, potrebno je dokazati da se pohlepnim algoritmom dobija korektno rešenje, što u nekim slučajevima može biti veoma izazovno. Samo nalaženje ispravnog pohlepnog algoritma može predstavljati ozbiljan problem i često nije trivijalno odrediti da li za neki problem postoji ili ne postoji pohlepno rešenje. Gramzivi algoritmi unapred znaju koja mogućnost će voditi do optimalnog rešenja i izbor vrše odmah, nakon čega rešavaju samo jedan potproblem. Potrebno je da važi svojstvo optimalne podstrukture tj. da se optimalno rešenje polaznog problema dobija pomoću optimalnog rešenja potproblema. Da bi se dokazala korektnost pohlepnog algoritma, obično je potrebno dokazati nekoliko stvari. Prvo je potrebno dokazati da strategija daje rešenje koje je ispravno. Nakon toga je potrebno dokazati i da je rešenje dobijeno strategijom optimalno. Obično se krene od nekog rešenja za koje pretpostavljamo da je optimalno i koje ne mora biti identično onome koje smo dobili pohlepnom strategijom. Ono ne može biti gore od rešenja nađenog na osnovu pohlepne strategije (jer ona vraća jedno korektno rešenje, pa optimum može biti samo eventualno bolji od tog rešenja), a potrebno je dokazati da ne može biti bolje. Jedna tehnika da se optimalnost dokaže je to da se pokaže da se optimalno rešenje može malo po malo, primenom transformacije pojedinačnih koraka, pretvoriti u rešenje dobijeno na osnovu naše strategije. Obično je dovoljno dokazati da se prvi korak optimalnog rešenja može zameniti prvim korakom koji gramziva strategija sugeriše, tako da se korektnost i kvalitet rešenja time ne narušavaju i korektnost dalje sledi na osnovu induktivnog argumenta. Ovu tehniku nazivaćemo *tehnikom razmene*. Druga tehnika da se optimalnost dokaže je to da se dokaže da je rešenje dobijeno na osnovu pohlepne strategije uvek po nekom kriterijumu ispred pretpostavljenog optimalnog rešenja. Ovu tehniku nazivaćemo *pohlepno rešenje je uvek ispred*. Treća tehnika da se optimalnost dokaže je da se odredi teorijska granica vrednosti optimuma i da se onda dokaže da pohlepni algoritam daje rešenje čija je vrednost upravo jednaka optimumu. Ovu tehniku nazivaćemo *tehnikom granice*.

11.2 Pohlepni algoritmi - raspored sa najviše aktivnosti

Problem: U jednom kabinetu se subotom održava obuka programiranja. Svaki nastavnik drži jedno predavanje i napisao je vreme u kom želi da drži nastavu (poznat je sat i minut početka i sat i minut završetka predavanja). Odredi kako je moguće napraviti raspored časova tako da što više nastavnika bude uključeno. Napisati program koji određuje optimalan raspored.

Dakle, pretpostavljamo da nam je dat niz od n intervala oblika $[s_i, f_i)$. Dva intervala $[s_i, f_i)$ i $[s_j, f_j)$ su kompatibilna ako im je presek prazan tj. ako je ili $f_i \leq s_j$ ili je $f_j \leq s_i$. Potrebno je pronaći kardinalnost maksimalnog podskupa međusobno kompatibilnih intervala.
Jedan način da se problem reši je da se ispitaju svi mogući podskupovi skupa časova, odaberu oni u kojima se časovi ne preklapaju i među njima pronađu oni koji sadrže maksimalni broj nastavnika. Složenost ovog pristupa bila bi ekspoencijalna i jasno je da on ne bi mogao da se u praksi primeni na probleme koji imaju više od nekoliko desetina predavanja.
Optimalno rešenje se dobija na osnovu intuicije koja nam govori da je dobro prvo zakazati onaj čas nakon čijeg održavanja učionica ostaje što duže slobodna, tj. od svih časova prvo zakazati onaj čas koji se najranije od svih časova završava. Nakon toga, potrebno je iz skupa izbaciti taj čas i sve one časove koji se sa njim preklapaju i nastaviti rekurzivno rešavanje problema sve dok se skup potencijalnih časova ne isprazni.

```
typedef pair<int, int> cas;
cas napraviCas(int pocSat, int pocMin, int krajSat, int krajMin) {
    return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}
inline int pocetakCasa(const cas& c) {
    return c.first;
}
inline int krajCasa(const cas& c) {
    return c.second;
}

int main() {
    int n; cin >> n;
    vector<cas> casovi(n);
    for (int i = 0; i < n; i++) {
        int pocSat, pocMin, krajSat, krajMin;
        cin >> pocSat >> pocMin >> krajSat >> krajMin;
        casovi[i] = napraviCas(pocSat, pocMin, krajSat, krajMin);
    }
    sort(begin(casovi), end(casovi), [](const cas& a, const cas& b) {
        return krajCasa(a) < krajCasa(b);
    });
    int brojOdrzanihCasova = 1;
    int kraj = krajCasa(casovi[0]);
    for (int i = 1; i < n; i++)
        if (pocetakCasa(casovi[i]) >= kraj) {
            brojOdrzanihCasova++;
            kraj = krajCasa(casovi[i]);
        }
    cout << brojOdrzanihCasova << endl;
    return 0;
}
```

Dokažimo optimalnost korišćenjem tehnike razmene. Pretpostavimo da je $O = \{c_1, ..., c_k\}$, skup časova koji predstavlja neko optimalno rešenje, pri čemu su časovi c_1 do c_k sortirani neopadajuće po redosledu njihovog završetka. Pošto se svi ti časovi mogu održati, između njih nema preklapanja i svaki naredni počinje nakon završetka prethodnog. Neka je $c_i = [s_i, f_i)$ prvi čas u ovom skupu koji ne bi bio izabran našom strategijom. Pretpostavimo da bi naša strategija umesto njega odabrala čas $c'_i = [s'_i, f'_i)$. Pokažimo da se zamenom časa c_i časom c'_i dobija takođe raspored koji je optimalan. Na osnovu definicije strategije, čas c'_i se bira između časova koji počinju posle časa c_{i-1}, pa je $s'_i \geq f_{i-1}$ i taj čas je kompatibilan sa svim ranije održanim časovima. Potrebno je da pokažemo da je kompatibilan i sa svim časovima koji se održavaju kasnije. Pokažimo da se c'_i završava pre c_i (ili se eventualno završavaju istovremeno), tj. da je $f'_i \leq f_i$. Zaista, ako je i = 0, tada naša strategija bira c'_i koji se prvi završava, pa se stoga c_i ne može završavati pre njega. Ako je i > 0, tada naša strategija bira onaj c'_i koji se najranije završava iz skupa svih časova koji koji počinju nakon c_{i-1}. Pošto je početni raspored korektan, znamo da c_i mora pripadati tom skupu. Zato znamo da se c'_i mora završiti pre c_i. Dakle, ako postoje časovi u O pre časa c_i, oni ostaju nepromenjeni i čas c'_i se ne preklapa sa njima. Pošto se c'_i ne završava kasnije nego c_i, on se sigurno ne preklapa ni sa jednim časom iz O koji ide posle c_i. Dakle, kada se c_i zameni sa c'_i i dalje se dobija ispravan raspored sa istim brojem održanih časova kao O za koji smo pretpostavili da je optimalan. Po istom prinicpu možemo menjati naredne časove (ovo se mora zaustaviti jer u svakom narednom optimalnom skupu imamo po jedan čas više koji je u skladu sa našom strategijom) i tako pokazati da će naša strategija vratiti optimalan skup.

11.3 Pohlepni algoritmi - raspored sa najmanjim brojem učionica

Problem: Za svaki od n časova poznato je vreme početka i završetka. Napiši program koji određuje minimalni broj učionica potreban da se svi časovi održe. Pošto se zahteva da se svi časovi održe, obilazićemo ih u određenom redosledu i svaki čas ćemo pridruživati nekoj od slobodnih učionica. U trenutku u kom nema više slobodnih učionica koje su ranije otvorene, otvaraćemo novu učionicu. Znamo da je najmanji broj učionica sigurno veći ili jednak najvećem broju časova koji se istovremeno održavaju u nekom trenutku. U nastavku ćemo dokazati da je minimalan broj učionica uvek jednak tom broju i da se raspored može napraviti ako se časovi obilaze u rastućem redosledu njihovog početka.
Dokažimo da je naša strategija optimalna. Strategija je takva da je jedini razlog da se nova učionica otvori to da su sve ranije otvorene učionice već popunjene, tj. da postoji neki čas (recimo $[s_j, f_j)$) koji se seče sa svim časovima koji su raspoređeni u trenutnih d otvorenih učionica. Pošto su časovi sortirani na osnovu vremena početka, svih tih d časova počinje pre trenutka s_j i završava se nakon trenutka s_j (jer traju u trenutku s_j). To znači da u trenutku s_j sigurno postoji $d + 1$ časova koji se u tom trenutku održavaju, pa broj učionica mora biti bar $d+1$.

Dakle, ako se na osnovu strategije rezerviše nova učionica, sigurni smo da je to neophodno. Ako je naša strategija napravila raspored u nekom broju učionica, sigurni smo da nije bilo moguće napraviti raspored u manjem broju učionica, što znači da je napravljeni raspored optimalan.
Prvi korak u implementaciji je veoma jednostavan - učitavamo sve časove u niz i sortiramo ih na osnovu početnog vremena. Ključni korak u drugoj fazi je određivanje učionice u koju može biti smešten tekući čas. Za sve do tada otvorene učionice znamo vremena završetka časova u njima. Možemo pronaći učionicu u kojoj se čas najranije završava i proveriti da li je moguće da u nju rasporedimo tekući čas. Ako jeste, njoj ažuriramo vreme završetka časa, a ako nije, onda moramo otvoriti novu učionicu. Da bismo efikasno mogli da nađemo učionicu u kojoj se čas najranije završava, sve učionice možemo čuvati u redu sa prioritetom sortiranom po vremenu završetka časa u svakoj od učionica. Ako taj red nije prazan i ako je vreme završetka časa u učionici na vrhu reda manje ili jednako vremenu početka tekućeg časa, vreme završetka časa u toj učionici ažuriramo na vreme završetka tekućeg časa. U suprotnom u red dodajemo novu učionicu kojoj je vreme završetka časa postavljeno na vreme završetka tekućeg časa.
Ukupna složenost algoritma je $O(nlogn)$ i u fazi sortiranja i u fazi raspoređivanja.

```
struct Cas {
    int broj, pocetak, kraj;
};

Cas napraviCas(int broj, int pocSat, int pocMin, int krajSat,
int krajMin) {
    Cas c;
    c.broj = broj;
    c.pocetak = pocSat*60 + pocMin;
    c.kraj = krajSat*60 + krajMin;
    return c;
}

struct Ucionica {
    int broj;
    int slobodnaOd;
};

Ucionica napraviUcionicu(int slobodnaOd, int broj) {
    Ucionica u;
    u.broj = broj;
    u.slobodnaOd = slobodnaOd;
    return u;
}

struct PorediUcionice {
    bool operator()(const Ucionica& u1, const Ucionica& u2) {
```

```
        return u1.slobodnaOd > u2.slobodnaOd;
    }
};

int main() {
    int n;
    cin >> n;
    vector<Cas> casovi(n);

    for (int i = 0; i < n; i++) {
        int pocSat, pocMin, krajSat, krajMin;
        cin >> pocSat >> pocMin >> krajSat >> krajMin;
        casovi[i] = napraviCas(i, pocSat, pocMin, krajSat, krajMin);
    }

    sort(begin(casovi), end(casovi),
      [](const Cas& c1, const Cas& c2) {
        return c1.pocetak < c2.pocetak;
    });

    vector<int> ucionica(n);
    priority_queue<Ucionica, vector<Ucionica>, PorediUcionice>
        redUcionica;

    for (const Cas& c : casovi) {
        int brojUcionice;
        if (redUcionica.empty() ||
            redUcionica.top().slobodnaOd > c.pocetak)
              brojUcionice = redUcionica.size() + 1;
        else {
            brojUcionice = redUcionica.top().broj;
            redUcionica.pop();
        }
        ucionica[c.broj] = brojUcionice;
        redUcionica.push(napraviUcionicu(c.kraj, brojUcionice));
    }

    cout << redUcionica.size() << endl;
    for (int u : ucionica)
        cout << u << endl;

    return 0;
}
```

11.4 Pohlepni algoritmi - Hafmanovo kodiranje

Problem: Neka je zadat tekst koji je potrebno zapisati sa što manje bitova. Pritom je svaki znak teksta predstavljen jedinstvenim nizom bitova - kodom tj. kodnom rečju tog znaka. Ako su dužine kodova svih znakova jednake, broj bitova koji predstavljaju tekst zavisi samo od broja karaktera u njemu. Da bi se postigla ušteda, moraju se neki znaci kodirati manjim, a neki većim brojem bitova. Neka se u tekstu javlja n različitih karaktera i neka se karakter c_i javlja f_i puta i neka je za njegovo kodiranje potrebno b_i bitova. Jedan od uslova da se obezbedi jednoznačno dekodiranje je da nijedna kodna reč ne bude prefiks nekoj drugoj. Binarnim prefiksnim kodovima jednoznačno odgovaraju binarna drveta. Kôd svakog čvora se dobija obilaskom drveta - korak na levo dodaje simbol 0 na kod, a korak na desno simbol 1, pri čemu se karakteri koje je potrebno kodirati nalaze u listovima drveta. Za dati skup karaktera i frekvencije njihovog pojavljivanja konstruisati optimalni binarni prefiksni kôd.
Razmotrimo prvo strukturu drveta koje odgovara binarnom prefiksnom kodu. Prvi važan zaključak je da u optimalnom prefiksnom kodu svi unutrašnji čvorovi moraju imati oba deteta. U suprotnom se kôd može skratiti tako što se unutrašnji čvor ukloni i zameni svojom decom.
Intuicija nam govori da je poželjno karakterima koji se pojavljuju često dodeljivati kraće kodove. Zato će karakteri koji se javljaju ređe imati duže kodove. Naredno važno tvrđenje je to da se dva karaktera koji se najređe javljaju u optimalnom drvetu mogu naći kao dva susedna lista najudaljenija od korena. Dakle, znamo da postoji optimalni binarni prefiksni kôd u kome su dva najređa karaktera c_i i c_j deca istog čvora i dva najudaljenija lista. Njihovim uklanjajem iz optimalnog drveta A za polaznu azbuku dobija se drvo B za koje tvrdimo da je optimalno za azbuku u kojoj su ta dva karaktera uklonjena i zamenjena sa novim karakterom c čija je frekvencija $f_i + f_j$.
Na osnovu prethodnog razmatranja lako se može formulisati induktivno- rekurzivni algoritam. Određujemo dva karaktera sa najmanjim frekvencijama pojavljivanja, menjamo ih novim karakterom čija je frekvencija jednaka zbiru njihovih frekvencija, rekurzivno konstruišemo optimalno drvo i na kraju u tom drvetu na list koji odgovara novom karakteru dopisujemo dva nova lista koji odgovaraju uklonjenim karakterima sa najmanjim frekvencijama. Bazu indukcije predstavlja slučaj kada ostanu samo dva karaktera (i tada i jedan i drugi kodiramo sa po jednim bitom, tj. kreiramo koren drveta čija su ta dva karaktera listovi).

```
void procitajKodove(const vector<int>& levo,
  const vector<int>& desno, const vector<char>& karakteri, int i,
  int n, const string& kod, map<char, string>& kodovi) {

    if (i < n)
        kodovi[karakteri[i]] = kod;
    else {
        procitajKodove(levo, desno, karakteri, levo[i-n], n,
                       kod + "0", kodovi);
```

```
        procitajKodove(levo, desno, karakteri, desno[i-n], n,
                       kod + "1", kodovi);
    }
}

void procitajKodove(const vector<int>& levo,
  const vector<int>& desno, const vector<char>& karakteri, int n,
  map<char, string>& kodovi) {
    procitajKodove(levo, desno, karakteri, 2*n-2, n, "", kodovi);
}

int brojBitova(const vector<char>& karakteri,
const vector<int>& frekvencije) {
    int n = karakteri.size();
    vector<int> levo(n-1), desno(n-1);
    priority_queue<pair<int, int>, vector<pair<int, int>>,
                   greater<pair<int, int>>> pq;

    for (int i = 0; i < n; i++)
        pq.push(make_pair(frekvencije[i], i));

    for (int i = n; i < 2*n - 1; i++) {
        auto f1 = pq.top();
        pq.pop();
        auto f2 = pq.top();
        pq.pop();
        levo[i - n] = f1.second; desno[i - n] = f2.second;
        pq.push(make_pair(f1.first + f2.first, i));
    }

    map<char, string> kodovi;
    procitajKodove(levo, desno, karakteri, n, kodovi);
    map<char, int> frekvencijeKaraktera;

    for (int i = 0; i < n; i++)
        frekvencijeKaraktera[karakteri[i]] = frekvencije[i];

    int brojBitova = 0;
    for (auto it : kodovi)
        brojBitova += frekvencijeKaraktera[it.first] *
            it.second.length();

    return brojBitova;
}
```

11.5 Pohlepni algoritmi - plesni parovi

Problem: Poznate su visine n momaka i n devojaka. Napisati program koji određuje koliko se najviše plesnih parova može formirati tako da je momak uvek viši od devojke.
Jedna mogućnost je da se parovi formiraju tako što se upari k najviših momaka sa k najnižih devojaka. Ta strategija bi bila korektna, ali njena implementacija nije trivijalna, jer nije jasno koliko maksimalno može da bude k. Varijacija koju ćemo jednostavno implementirati je sledeća. Ako postoji bar jedan plesni par, u njemu može da učestvuje najviši mladić. Naime, ako on ne bi učestvovao ni u jednom paru, uvek bismo nekog od mladića mogli zameniti njime i dobiti isti broj plesnih parova. Postavlja se pitanje sa kojom devojkom on treba da pleše. Cilj nam je da nakon formiranja tog para preostanu što niže devojke, da bi niži mladići imali šanse da se sa njima upare. Jasno je da moramo da eliminišemo sve devojke koje su više od tog najvišeg mladića, a od preostalih devojaka možemo da izaberemo najvišu. Nakon eliminisanja tog mladića, svih devojaka viših od njega i devojke sa kojom pleše, problem je sveden na problem istog oblika, ali manje dimenzije. Izlaz predstavlja slučaj kada su sve devojke više od najvišeg među preostalim mladićima.
Prilikom implementacije skup momaka i devojaka možemo čuvati u nizovima uređenim u opadajućem redosledu visine. Niz momaka obilazimo redom, element po element, a u niz devojaka razdvajamo na one koje su eliminisane. Održavamo mesto početka niza devojaka koje još nisu obrađene i prilikom traženja devojke za tekućeg momka niz devojaka obilazimo od te pozicije. Svaku devojku ili eliminišemo, jer je viša od tekućeg momka ili je dodeljujemo tekućem momku i onda ih oboje eliminišemo. Naglasimo da se u implementaciji ne moramo vraćati na eliminisane devojke, jer ako je neka devojka viša od tekućeg momka, biće viša i od svih narednih. Stoga se oba pokazivača kreću samo u jednom smeru i složenost faze dodeljivanja je linearna. Ukupnim algoritmom dominira složenost sortiranja, pa je ukupna složenost $O(nlogn)$.

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

    vector<int> momci(n);
    for (int i = 0; i < n; i++)
        cin >> moci[i];

    vector<int> devojke(n);
    for (int i = 0; i < n; i++)
        cin >> devojke[i];

    sort(begin(momci), end(momci), greater<int>());
    sort(begin(devojke), end(devojke), greater<int>());

    int brojParova = 0;
    int m = 0, d = 0;
```

```
    while (true) {
        while (d < n && momci[m] < devojke[d])
            d++;
        if (d >= n)
            break;
        brojParova++;
        m++;
        d++;
    }

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

11.6 Pohlepni algoritmi - razlomljeni problem ranca

Problem: U jednoj prodavnici se prodaju slatkiši "na meru". Postoji n vrsta slatkiša i znamo da i-tog slatkiša ima w_i grama, po ukupnoj ceni od v_i dinara. Prodavnica je u okviru svoje promocije organizovala nagradnu igru u kojoj je nagradila jednu svoju mušteriju tako da na poklon može da uzme sve slatkiše koji staju u ranac nosivosti W grama. Kolika je najveća vrednost slatkiša koje sretni dobitnik može da uzme.
Pošto sretni dobitnik želi da pokupi što veću vrednost slatkiša, jasno je da treba da krene uzimanje onih slatkiša koji su najvredniji. Ako tom vrstom slatkiša može da ispuni ceo ranac, najbolje mu je da to uradi (pretpostavljamo da je moguće da ne pokupi celokupnu raspoloživu količinu tog slatikša). U suprotnom, pokupiće celokupnu količinu tog slatkiša, a zatim će preostalu nosivost ranca popuniti preostalim slatkišima, po istom principu.

```
double razlomljeniRanac(const vector<int>& cena,
const vector<int>& kolicina, int nosivostRanca) {
    int n = cena.size();
    vector<pair<double, int>> jedinicnaCenaIKolicina(n);

    for (int i = 0; i < n; i++) {
        double jedinicnaCena = (double)cena[i] / (double)kolicina[i];
        jedinicnaCenaIKolicina[i] = make_pair(jedinicnaCena,
                                              kolicina[i]);
    }

    sort(begin(jedinicnaCenaIKolicina), end(jedinicnaCenaIKolicina),
         greater<pair<double, int>>());
    double ukupnaVrednost = 0.0;

    for (int i = 0; nosivostRanca > 0 && i < n; i++) {
        double jedinicnaCena = jedinicnaCenaIKolicina[i].first;
        int kolicina = jedinicnaCenaIKolicina[i].second;
```

```
            int uzetaKolicina = min(kolicina, nosivostRanca);
            nosivostRanca -= uzetaKolicina;
            ukupnaVrednost += uzetaKolicina * jedinicnaCena;
        }

        return ukupnaVrednost;
    }
```

Jasno je da prethodni gramzivi algoritam daje uvek korektno rešenje, jer se ni za jedan predmet ne uzima veća količina od dostupne i ukupna masa uzetih slatkiša ne prevazilazi masu ranca.