Čas 1.1 - Uvod

Obaveze studenata

- Predispitne obaveze:
  - test sa vežbi - 20 poena
  - test sa predavanja - 10 poena
- Završni ispit:
  - praktični ispit - 30 poena (prag 15 poena)
  - usmeni ispit - 40 poena (prag 20 poena)

Literatura

- Filip Marić, Vesna Marinković, Mladen Nikolić: *Algoritmi i strukture podataka*, beleške sa predavanja i vežbi
- Miodrag Živković: *Algoritmi*
- Miodrag Živkoković, Vesna Marinković: *Algoritmi i strukture podataka*, skripta
- Predrag Janičić, Filip Marić: *Programiranje 2*, skripta
- Thomas H. Cormen. Charles E. Leiserson. Ronald L. Rivest. Clifford Stein. *Introduction to Algorithms*. Third Edition. The MIT Press.
- Steven S. Skiena. *The Algorithm Design Manual*, Springer.
- onlajn literatura (petlja.org, geeksforgeeks.org, …)

Preduslovi

- Nema formalnih preduslova, ali se podrazumeva da ste savladali u potpunosti gradivo koje se predaje na P1/P2, kao i elementarne pojmove diskretne matematike (koji se pokrivaju i u srednjim školama). Potrebno je, na primer, da umete da rešite sve naredne zadatke.
  - Navesti bar jedan algoritam čije vreme izvršavanja zadovoljava rekurentnu jednačinu $T(n) = 2T(n/2) + O(n)$, $T(1) = O(1)$. Koja je rezultujuća složenost?
  - Opisati i implementirati korak particionisanja kod algoritma QuickSort.
  - Definisati strukturu čvora dvostruko povezane liste i funkciju koja briše takvu listu (poznat je pokazivač na njen prvi član).
  - Navesti primer niza od k elemenata za koji se algoritam umetanja jednog po jednog čvora u nebalansirano pretraživačko binarno stablo izvršava najsporije. Koja je složenost umetanja svih elemenata u tom slučaju?

– Definisati rekurzivnu funkciju koja izračunava $1 + ... + n$. Kolika je vrednost tog zbira?

– Indukcijom dokaži da je $1 + 3 + ... + (2k + 1) = (k + 1)^2$.

– Ukloni rekurziju iz naredne funkcije koja određuje koliko se kvadrata može iseći od pravougaonika dimenzije $a \times b$, ako se u svakom koraku iseca kvadrat manje dimenzije.

```
int brojKvadrata(int a, int b) {
   if (b == 0)
      return 0;
   return a / b + brojKvadrata(b, a % b);
}
```

Pojam algoritma i strukture podataka

Kao što sam naziv kursa govori, ključne teme ovog kursa predstavljaju algoritmi i strukture podataka.

- Niklaus Wirth (tvorac Pascal-a): **algoritmi + strukture podataka = programi**
- Algoritmi su sredstva za rešavanje precizno definisanih računskih problema. Npr. problem neopadajućeg sortiranja niza brojeva se može definisati na sledeći način:
  - Ulaz: niz brojeva $(a_0, a_1, ..., a_{n-1})$
  - Izlaz: permutacija $(a'_0, a'_1, ..., a'_{n-1})$ ulaznog niza takva da je $a'_0 \leq a'_1 \leq ... \leq a'_{n-1}$.
- Veoma 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 između njih i funkcije i operacije koje se mogu primeniti nad tim podacima.

Načini opisa algoritama

Neki od najčešće korišćenih mehanizama za opis algoritama su sledeći:

- programski kôd
- pseudo-kôd
- blok dijagrami toka

- scratch/blockly dijagrami
- ...

Formalne definicije algoritama

Postoje precizne definicije pojma algoritma (nastale 1930-ih kada su se logičari bavili pitanjem šta se može, a šta ne može izračunati praćenjem precizno definisanih postupaka).

- URM mašine
- Tjuringove mašine
- Gedelove rekurzivne funkcije
- Čerčov lambda račun
- ...

Čas 1.2, 1.3 - Korektnost algoritama i veza sa induktivno/rekurzivnom konstrukcijom

Korektnost algoritma je osobina bez koje se ne može. Korektnost se ogleda kroz dva aspekta.

Parcijalna koretnost: Vrednost koju algoritam vrati mora da ispunjava specifikaciju zadatka.

Zaustavljanje: Algoritam mora da se zaustavi za sve ulaze za koje želimo da izračunamo vrednost.

Većina algoritama koje ćemo proučavati u ovom kursu biće potpuni tj. zaustavljaće se za sve dopuštene ulaze. Za zaustavljajuće parcijalno korektne algoritme kažemo da su *totalno korektni*. Interesantno, dokazano je da ne postoje algoritmi kojima bi se ispitivala gore navedena svojstva algoritama.

Rekurzija i matematička indukcija, iteracija i invarijanta petlje

Ključna ideja ovog kursa je to da je konstrukcija algoritama veoma tesno povezana sa dokazivanjem teorema matematičkom indukcijom!

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 svođenja na

probleme manje dimenzije. Ako se prilikom svođenja dimenzija problema uvek smanjuje, konstruisani algoritmi će se uvek zaustavljati. Implementacija ovako dizajniranih algoritama može, ali ne mora biti isključivo rekurzivna.

U nastavku ovog poglavlja pokazaćemo da je definisanje algoritama u veoma tesnoj vezi sa dokazivanjem njihove korektnosti. Iako postoje formalni okviri za dokazivanje korektnosti imperativnih programa (pre svega Horova logika), mi ćemo se baviti isključivo neformalnim dokazima i veza između logike u kojoj vršimo dokazivanje i (imperativnog) programskog jezika u kojem se program izražava biće prilično neformalna. Ipak, skrenimo pažnju na to da promenljive u matematici i u programiranju imaju različite osobine. Naime, promenljive u matematici označavaju (imenuju) jednu vrednost dok promenljive u (imperativnom) programiraju imaju dinamički karakter i menjaju svoje vrednosti tokom izvršavanja programa. Na primer, brojačka promenljiva `i` u nekoj petlji može redom imati vrednosti 1, 2 i 3. Da bismo napravili razliku između ove dve različite vrste promenljivih koristićemo različit font - promenljivu programa ćemo obeležavati sa `i`, a njenu vrednost sa i. Ako želimo da razlikujemo staru i novu vrednost promenljive `i`, koristićemo oznake i i i'. Ako želimo da naglasimo da je promenljiva redom uzimala neku seriju vrednosti, koristićemo oznake i_0 (početna vrednost promenljive `i`), i_1, i_2, … U situacijama u kojima se vrednost promenljive ne menja (na primer, ako je dužina niza tokom celog trajanja programa ista), nećemo obraćati pažnju na razliku između promenljive programa (npr. `n`) i njene vrednosti (npr. n). Elemente nizova ćemo takođe obeležavati indeksima i obično ćemo pretpostaljati da brojanje kreće od nule (npr. a_0, a_1, …).

Recimo i da ćemo prilikom dokazivanja korektnosti programa obično ignorisati ograničenja zapisa brojeva u računaru i podrazumevaćemo da je opseg brojeva neograničen i da se realni brojevi zapisuju sa maksimalnom preciznošću. Dakle, nećemo obraćati pažnju na greške koje mogu nastati usled prekoračenja ili potkoračenja vrednosti tokom izvođenja aritmetičkih operacija.

Određivanja minimuma niza

Problem: Definisati funkciju koja određuje minimum nepraznog niza brojeva i dokaži njenu korektnost. Razmotriti rekurzivnu i iterativnu implementaciju.

Pokažimo kako se algoritam jednostavno konstruiše tzv. induktivno-rekurzivnom konstrukcijom. Dimenzija problema u ovom slučaju je broj elemenata niza.

Baza: Ako niz ima samo jedan element, tada je taj element ujedno i minimum.

Korak: U suprotnom, pretpostavimo da nekako umemo da rešimo problem za manju dimenziju i na osnovu toga pokušajmo da dobijemo rešenje za ceo niz. Dakle, pretpostavimo da je dužina niza $n > 1$ i da umemo da nađemo broj m koji predstavlja minimum prvih $n - 1$ elemenata niza. Minimum

celog niza dužine n je manji od brojeva m i preostalog, n-tog elementa niza (ako brojanje kreće od 0, to je element a_{n-1}).

Ovu konstrukciju lako možemo pretočiti u rekurzivnu funkciju.

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

int nadji_manji(int a, int b) {
    return a < b ? a : b;
}

int min(int a[], int n) {
    if (n == 1)
       return a[0];
    else {
       int m = min(a, n-1);
       return nadji_manji(m, a[n-1]);
    }
}

int main() {
    int a[] = {3, 5, 4, 1, 6, 2, 7};
    int n = sizeof(a) / sizeof(int);
    cout << min(a, n) << endl;
}
```

Recimo i da smo umesto definisanja funkcije `min2` za određivanje minimuma dva broja mogli koristiti i funkciju `std::min` iz zaglavlja `<algorithm>`. Ipak, u ovim prvim programima nećemo koristiti napredne mogućnosti jezika C++, da bismo naglasili da tehnike koje u ovom poglavlju uvodimo nisu ni po čemu specifične za taj jezik.

Korektnost prethodnog algortima se može formulisati u obliku sledeće teoreme.

Teorema: Za svaki neprazan niz a (niz za koji je dužina $|a| \geq 1$) i za svako $1 \leq n \leq |a|$ poziv `min(a, n)` vraća najmanji među prvih n elementa niza a (sa a i n su obeležene vrednosti niza `a` i promenljive `n`, a sa $|a|$ dužina niza `a`).

Tu teoremu možemo dokazati indukcijom.

- Bazu indukcije predstavlja slučaj $n = 1$, 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 (jer je on ujedno najmanji među prvih 1 elemenata niza).
- Kao induktivnu hipotezu možemo pretpostaviti da ako važi $1 \leq n-1 < |a|$, tada poziv `min(a, n-1)` vraća najmanji od prvih $n-1$ elemenata niza a. Iz te pretpostavke potrebno je da dokažemo da za n koje zadovoljava $1 < n \leq |a|$ poziv `min(a, n)` vraća najmanji od prvih n elemenata niza a (pri tom je a neprazan niz). Na osnovu definicije funkcije `min`, poziv `min(a, n)` će vratiti manji od brojeva m (koji predstavlja rezultat poziva

`min(a, n-1)`) i a_{n-1}. Pošto su uslovi induktivne hipoteze zadovoljeni, 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 (elementa a_{n-1}) biti najmanji među prvih n elemenata niza a.

Primećujemo ogromnu sličnost između rekurzivne konstrukcije algoritma i induktivnog dokaza njegove korektnosti. Stoga slobodno možemo da kažemo da su rekurzija i indukcija dve strane jedne te iste medalje (indukciju koristimo kao tehniku dokazivanja, a rekurziju kao tehniku konstrukcije algoritama). U nastavku ćemo termine rekrzivno i induktivno koristiti često kao sinonime, a tehniku konstrukcije algoritama zvaćemo induktivno-rekurzivnom.

Recimo i da je ovaj oblik korišćenja matematičke indukcije malo nestandardan, jer se ne koristi direktno indukcija po prirodnim brojevima, već se koristi indukcija po strukturi rekurzivne funkcije u kojoj se, iz pretpostavke da svaki rekurzivni poziv vraća korektan rezultat, dokazuje da funkcija vraća korektan rezultat. Takva teorema indukcije se može dokazati na osnovu klasične matematičke indukcije po broju rekurzivnih poziva, pod pretpostavkom da se dokaže da se rekurzivna funkcija uvek zaustavlja.

Induktivno/rekurzivnu konstrukciju možemo pretočiti i u iterativni algoritam.

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

int min2(int a, int b) {
    return a < b ? a : b;
}

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++)
        m = min2(m, a[i]);
    cout << m << endl;
}
```

I u ovom rešenju se može prepoznati identična induktivna konstrukcija koja je upotrebljena u rekurzivnom rešenju. Naravno, možemo eliminisati funkciju za izračunavanje minimuma dva broja.

```
    int m = a[0];
    for (int i = 1; i < n; i++)
        if (a[i] < m)
           m = a[i];
```

U svakom koraku petlje niz čiji minimum znamo postaje duži za po jedan element. Možemo dokazati sledeću teoremu.

Teorema: Ako je niz a dužine $n \geq 1$, u svakom koraku petlje (i na njenom početku, neposredno nakon provere uslova, ali i na njenom kraju, neposredno nakon izvršavanja koraka) važi da je $1 \leq i \leq n$ i da je m minimum prvih i elemenata niza (gde je i tekuća vrednost promenljive `i`, a m tekuća vrednost promenljive `m`).

I ovo tvrđenje možemo dokazati indukcijom i to po broju izvršavanja tela petlje (obeležimo taj broj sa k). Napomenimo samo da ćemo petlju `for` smatrati samo skraćenicom za petlju `while`, tako da ćemo inicijalizaciju petlje smatrati za kôd koji se izvršava pre petlje, dok ćemo korak petlje smatrati kao poslednju naredbu tela petlje. Takođe, implicitno ćemo podrazumevati da se tokom izvršavanja petlje niz ni u jednom trenutku ne menja (i to se eksplicitno može dokazati indukcijom).

- Bazu indukcije čini slučaj $k = 0$ tj. slučaj kada se telo petlje nije još izvršilo. 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 sada kao induktivnu hipotezu da tvrđenje važi nakon k izvršavanja tela petlje, tj. da je m_k (vrednost promenljive `m` nakon k izvršavanja tela petlje) jednako minimumu prvih i_k elemenata niza (gde je i_k vrednost brojačke promenljive `i` nakon k izvršavanja tela petlje). Nakon k izvršavanja tela petlje promenljiva i_k će imati vrednost $k + 1$ (jer je imala početnu vrednost 1 i tačno k puta je uvećana za 1 - i ovo bi se formalno moglo dokazati indukcijom), tako da induktivna hipoteza tvrdi da promenljiva `m` sadrži vrednost minimuma prvih $k + 1$ članova niza. Iz te pretpostavke i pretpostavke da je uslov petlje ispunjen (tj. da je `i < n` odnosno da je $i_k = k + 1 < n$) dokažimo da nakon $k + 1$ izrvršavanja tela petlje važi da je m_{k+1} (vrednost promenljiva `m` nakon $k + 1$ izvršavanja tela petlje) minimum prvih i_{k+1} elementa niza (sa i_{k+1} označili vrednost promenljive `i` nakon $k+1$ izvršavanja tela petlje, a m_{k+1} vrednost). Vrednosti promenljivih m_{k+1} i i_{k+1} se mogu lako odrediti na osnovu vrednosti m_k i i_k, analizom jednog izvršavanja tela (i koraka) petlje. Važi da je $i_{k+1} = i_k + 1 = k + 2$, dok je m_{k+1} minimum vrednosti m_k i elementa a_{i_k}, tj. a_{k+1}. Na osnovu induktivne hipoteze znamo da je m_k minimum prvih $i_k = k + 1$ elemenata niza. Zato će m_{k+1} biti minimum prvih $k + 2$ elemenata niza (zaključno sa elementom a_{k+1}), što je tačno i_{k+1} elemenata niza. Pošto je $i_k < n$ važi i da je $i_{k+1} \leq n$.

Kada se petlja završi važi da je $i = n$ (jer znamo da je $i \leq n$, a uslov petlje $i < n$ nije ispunjen), tako da na osnovu upravo dokazanog tvrđenja znamo da promenljiva `m` sadrži minimum n članova niza (što je zapravo ceo niz, jer je n njegova dužina).

Ako razmotrimo strukturu prethodnog razmatranja, možemo ustanoviti da smo identifikovali logički uslov koji je ispunjen neposredno pre i neposredno nakon svakog izvršavanja tela petlje. Takav uslov naziva se *invarijanta petlje*. Da

bismo dokazali da je neki uslov invarijanta petlje indukcijom na osnovu broja izvršavanja tela petlje, dovoljno je da dokažemo:

1. da taj uslov važi pre prvog ulaska u petlju i
2. 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.

Izolujmo ključne delove prethodnog dokaza i prikažimo ih u formatu koji ćemo ubuduće koristiti prilikom dokazivanja invarijanti petlji (indukcija će u tim dokazima biti samo implicitna).

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.

- 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` (označimo je sa m) jednaka minimumu prvih i članova niza (gde je i vrednost promenljive `i` na ulasku u petlju), kao i da je uslov petlje ispunjen tj. da je $i < n$. Nakon izvršavanja tela petlje nova vrednost promenljive `m` (označimo je sa m') biće jednaka manjoj od vrednosti m i a_i. Pošto je na osnovu pretpostavke m jednak minimumu prvih i elemenata niza, tj. minimumu brojeva a_0, …, a_{i-1}, pa je m' 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 (obeležimo novu vrednost sa i') 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.

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.

Svaka petlja ima puno invarijanti, međutim, od interesa su nam samo one invarijante koje u kombinaciji sa uslovom prekida petlje impliciraju uslov koji nam je potreban nakon petlje. Ako je petlja jedina u nekom algoritmu, obično je to onda uslov korektnosti samog algoritma. U tekućem primeru uslov da promenljiva m nakon k izvršavanja sadrži vrednost prvih $k + 1$ elemenata niza

zajedno sa uslovom da se petlja izvršava tačno $n-1$ puta garantuju da će nakon petlje promenljiva m da sadrži minimum celog n-točlanog niza.

Euklidov algoritam

Problem: Definisati funkciju koja izračunava najveći zajednički delilac dva neoznačena broja i dokazati njenu korektnost. Razmotriti rekurzivnu i iterativnu implementaciju.

Izračunavanje najvećeg zajedničkog delioca dva nenegativna broja a i b, može se izvršiti korišćenjem Euklidovog algoritma. Označimo sa $a \operatorname{div} b$ i $a \bmod b$ redom celobrojni količnik i ostatak pri deljenju brojeva a i b (ako je $a = b \cdot q + r$, tako da je $0 \leq r < b$, tada je $a \operatorname{div} b = q$ i $a \bmod b = r$). Algoritam razmatra dva slučaja:

Baza: ako je $b = 0$ tada je $nzd(a, 0) = a$,

Korak: za $b > 0$ važi: $nzd(a, b) = nzd(b, a \bmod b)$.

Na primer $nzd(18, 46) = nzd(46, 18) = nzd(18, 10) = nzd(10, 8) = nzd(8, 2) = nzd(2, 0) = 2$.

Postupak se uvek zaustavlja jer je vrednost $a \bmod b$ uvek nenegativna (i celobrojna) i smanjuje se, pa mora nekada doći do nule.

Ovu konstrukciju lako možemo pretočiti u rekurzivnu funkciju.

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

Korektnost algoritma može se dokazati indukcijom.

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

- Važi da je $nzd(a, 0) = a$, jer je a najveći broj koji deli i a i 0. Pošto kada promenljiva `b` ima vrednost 0 funkcija `nzd` vraća vrednost a promenljive `a`, tvrđenje važi.
- Pretpostavimo, kao induktivnu hipotezu, da poziv `nzd(b, a % b)` vraća najveći broj n koji deli broj b i broj $a \bmod b$. Dokažimo da je taj broj ujedno i $nzd(a, b)$. Važi da je $a = b \cdot q + a \bmod b$. Pošto n deli i b i $a \bmod b$, 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 $a = b \cdot q + a \bmod b$ taj broj sigurno mora da deli i $a \bmod b$. Međutim, pošto je n najveći zajednički delilac brojeva b i $a \bmod b$, 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)`.

Algoritam možemo realizovati i iterativno (pošto je rekurzija repna, veoma jednostavno je se oslobađamo).

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

Dokažimo, vežbe radi, korektnost ovog algoritma korišćenjem naredne invarijante petlje.

Lema: U svakom koraku petlje važi da je $nzd(a, b) = nzd(a_0, b_0)$, gde su a_0 i b_0 vrednosti argumenata funkcije.

- 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 $b \neq 0$. Nakon izvršavanja tela petlje, nove vrednosti promenljivih a i b su $a' = b$ i $b' = a \bmod b$, pa je $nzd(a', b') = nzd(b, a \bmod b)$. Na osnovu dokaza ranije opisane matematičke teoreme znamo da je $nzd(b, a \bmod b) = nzd(a, b)$, a na osnovu induktivne hipoteze (pretpostavke da invarijanta važi na ulazu u petlju) da je $nzd(a, b) = nzd(a_0, b_0)$. Dakle, uslov jeste invarijanta petlje.

Teorema: Poziv `nzd(a, b)` vraća najveći zajednički delilac brojeva a i b.

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

Određivanje binarnih cifara u zapisu prirodnog broja

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

Problem se opet rešava induktivno-rekurzivnim pristupom.

- Nula se zapisuje jednom binarnom cifrom nula.
- Pretpostavimo da broj $\lfloor \frac{n}{2} \rfloor$ tj. broj $n \operatorname{div} 2$ umemo da zapišemo binarno. Tada se binarni zapis broja n može dobiti tako što se na taj zapis zdesna dopiše cifra $n \bmod 2$.

```
string binarnaCifra(unsigned n) {
   return n % 2 == 0 ? "0" : "1";
}

string binarniZapis(unsigned n) {
   if (n == 0) return "0";
   return binarniZapis(n / 2) + binarnaCifra(n % 2);
}
```

Primetimo da se na ovaj način uvek dobija zapis koji počinje vodećom nulom, što je matematički potpuno ispravno, ali je malo ružno. Jedan način da se to izbegne je da se u glavnoj funkciji slučaj nule posebno obradi, a da se prilikom izlaza iz rekurzije u slučaju nule vrati prazna niska.

```
string binarniZapis_(unsigned n) {
   if (n == 0)
      return "";
   return binarniZapis_(n / 2) + binarnaCifra(n % 2);
}
string binarniZapis(unsigned n) {
   if (n == 0)
      return "0";
   return binarniZapis_(n);
}
```

Za vežbu vam ostavljamo da indukcijom dokažete korektnost ove funkcije.

Umesto da koristimo niske karaktera i njihovo nadovezivanje (što može biti neefikasna operacija), možemo rezultat smestiti u niz istinitosnih vrednosti (podataka tipa `bool`).

Naredna funkcija zapisuje cifre broja `n` u niz `binarneCifre` počevši od pozicije `i`, pa unapred.

```
void binarniZapis(unsigned n, bool binarneCifre[], i) {
   if (n == 0) return;
   binarneCifre[i] = n % 2;
   binarniZapis(n / 2, binarneCifre, i+1);
}
```

U glavnoj funkciji pretpostavljamo da se u nizu binarnih cifara binarne cifre smeštaju tako da se na poziciju i smešta cifra težine 2^i (cifra jedinica je na mestu 0) i da je niz inicijalno popunjen svim vrednostima `false`. Takođe, u datom rešenju pretpostavljamo da važi $n < 2^{32}$.

```
// prevođenje
bool binarneCifre[32] = {false};
binarniZapis(n, binarneCifre, 0);
```

Pretočimo ovo u iterativni algoritam i dokažimo njegovu korektnost korišćenjem invarijante petlje.

```
// prevođenje
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.

- 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 $2^{i'} \cdot n' + b' = 2^{i+1} \cdot (n \operatorname{div} 2) + b + 2^i \cdot n \bmod 2 = 2^i \cdot (2 \cdot (n \operatorname{div} 2) + n \bmod 2) + b$. Na osnovu definicije celobrojnog deljenja važi da je $2 \cdot (n \operatorname{div} 2) + n \bmod 2 = n$, pa je vrednost prethodnog izraza jednaka $2^i \cdot n + b$, a na osnovu induktivne hipoteze (pretpostavke o tome da invarijanta važi na ulasku u telo petlje) znamo da je to jednako n_0.

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

Kako je po izlasku iz petlje $n = 0$, na osnovu invarijante važi da je $b = n_0$ tj. da niz sadrži binarni zapis polaznog broja.

Obrtanje redosleda cifara u zapisu prirodnog broja

Problem: Neka je rev funkcija koja prirodni broj n preslikava u broj koji se zapisuje istim ciframa kao broj n, ali u obratnom poretku. Npr. $\text{rev}(1234) = 4321$, $\text{rev}(100) = 1$. Podrazumevamo da je $\text{rev}(0) = 0$. Definisati iterativni algoritam koji određuje vrednost $\text{rev}(n)$ i dokaži njenu korektnost.

```
int r = 0;
while (n > 0) {
   r = 10*r + n % 10;
   n /= 10;
}
```

U ovom postupku se kombinuju dva algoritma - algoritam za određivanja cifara u dekadnom zapisu broja, zdesna nalevo (određivanjem ostatka pri celobrojnom deljenju) i Hornerova shema za određivanje vrednosti broja na osnovu njegovih cifara sleva nadesno.

Nakon i koraka petlje broj n sadrži celobrojni količnik pri deljenju početnog broja n_0 brojem 10^i, dok broj r sadrži broj koji se dobije obrtanjem poslednjih i cifara broja n.

Skrenimo pažnju na jedan suptilan detalj koji je bitan u potpuno preciznom, formalnom dokazivanju teorema, a često se zanemaruje u neformalnom razma-

tranju korektnosti programa korišćenjem tehnike invarijanti. Primetimo da se u ovom uslovu pominje broj izvršavanja tela petlje i, dok takva vrednost ne figuriše u samom programu. Ako invarijantu posmatramo samo kao odnos vrednosti promenljivih u programu, onda prethodni uslov striktno gledano nije invarijanta jer uključuje i vrednosti koje se ne javljaju u programu (početnu vrednost n_0 i broj izvršenih koraka petlje i). Da bismo tu nesaglasnost ispravili, program možemo dopuniti tzv. *fantomskim promenljivama* koje ne služe izračunavanjima koje program sprovodi, ali služe prilikom dokazivanja njegove korektnosti.

```
int n0;
cin >> n0;
int n = n0, i = 0, r = 0;
while (n > 0) {
   r = 10 * r + n % 10;
   n /= 10;
   i++;
}
```

Softver koji se koristi za formalno dokazivanje korektnosti programa često dopušta uvođenje fantomskih promenljivih i u stanju je da ih automatski izbaci prilikom generisanja izvršivog programa, kako ne bi smetale efikasnosti. Prilikom neformalnog rezonovanja o kodu ovakve fantomske promenljive ćemo često razmatrati samo implicitno. Sada možemo formulisati lemu koja uključuje i fantomske promenljive.

Napomenimo da se broj koji se obrće može završiti nulama, tako da na primer brojevi 123, 1230, 12300 svi imaju istu vrednost broja koji se zapisuje istim ciframa kao originalni broj ali u obrnutom poretku (321). Ali broj petlji koji će biti potreban da se ta vrednost izračuna će se razlikovati u tim situacijama (i biće jednak broju cifara broja n). Otuda je bitno primetiti da je broj završnih nula u zapisu broja bitan za predloženi algoritam, i figurisaće u formulaciji naredne leme.

Lema: Uslov da je $n_0 = n \cdot 10^i + \mathrm{rev}(r) \cdot 10^s$ (gde je s broj završnih nula u zapisu broja n_0) je invarijanta petlje. Pri tom je broj r jednak 0 dok je $i \leq s$ (ima 0 cifara u zapisu), odnosno ima tačno $i - s$ cifara u zapisu kada je $i > s$ i vodeća cifra mu je različita od nule. Važi i da je $n \geq 0$.

Razmatramo

- Pre prvog ulaska u petlju je $n = n_0$, $r = 0$ i $i = 0$, pa uslov očigledno važi.
- Pretpostavimo da tvrđenje važi na ulasku u petlju. Efekat tela petlje je da je $n' = n \operatorname{div} 10$, da je $r' = 10r + n \bmod 10$ i da je $i' = i + 1$. Potrebno je da dokažemo da vrednost $n' \cdot 10^{i'} + \mathrm{rev}(r') \cdot 10^s$ ostaje jednaka n_0. Nakon izvršavanja tela petlje, ona je jednaka $(n \operatorname{div} 10) \cdot 10^{i+1} + \mathrm{rev}(10r + n \bmod 10) \cdot 10^s$. Ovu vrednost dalje izračunavamo u zavisnosti od odnosa promenljivih i i s.

Ako je $i < s$, petlja skida nule sa kraja zapisa broja n, tada je $r = 0$ i $n \bmod 10 = 0$. Zato je prethodna vrednost jednaka $(n \operatorname{div} 10) \cdot 10^{i+1} = (10 \cdot (n \operatorname{div} 10) + n \bmod 10) \cdot 10^i = n \cdot 10^i$, što je na osnovu pretpostavke (jer je i u prethodnom koraku $r = 0$) jednako n_0.

Ako je $i = s$ petlja je stigla do prve cifre različite od nule, pa je još uvek $r = 0$, a $n \bmod 10 \neq 0$. Pošto je $n \bmod 10$ jednocifren broj važi i da je $\text{rev}(n \bmod 10) = n \bmod 10$. Pošto je $i = s$, vrednost izraza koji figuriše u invarijanti jednaka je $(n \operatorname{div} 10) \cdot 10^{i+1} + n \bmod 10 \cdot 10^i = (10 \cdot (n \operatorname{div} 10) + n \bmod 10) \cdot 10^i = n \cdot 10^i$, a to je na osnovu pretpostavke (jer je i u prethodnom koraku $r = 0$) jednako n_0. Važi i da je $r' = n \bmod 10$, pa je $0 < r' < 10$ i r' je jednocifren broj. Što znači da r ima tačno ($i' - s = i + 1 - i = 1$) jednu cifru i vodeća cifra mu je različita od nule).

Ako je $i > s$, pošto r ima tačno i - s cifara, tada važi da je $\text{rev}(10r + n \bmod 10) = (n \bmod 10) \cdot 10^{i-s} + \text{rev}(r)$. Preciznije, izraz sa leve strane označava broj koji se dobija obrtanjem redosleda cifara broja koji se od broja r dobija dopisivanjem cifre $n \bmod 10$ (ona može biti i nula) s desne strane. Što se tiče izraza sa desne strane, na osnovu invarijante znamo da je r broj koji ima tačno $i - s$ cifara i one odgovaraju stepenima od 10^0 do 10^{i-s-1}. Zato se cifra $n \bmod 10$ množenjem sa faktorom 10^{i-s} postavlja kao prva cifra iza koje slede sve cifre zapisa broja r u obrnutom redosledu. Dakle, levi i desni izraz imaju istu vrednost. Na osnovu toga, znamo da je vrednost izraza koji figuriše u invarijanti nakon izvršavanja tela jednaka $(n \operatorname{div} 10) \cdot 10^{i+1} + ((n \bmod 10) \cdot 10^{i-s} + \text{rev}(r)) \cdot 10^s = (10 \cdot (n \operatorname{div} 10) + n \bmod 10) \cdot 10^i + \text{rev}(r) \cdot 10^s = n \cdot 10^i + \text{rev}(r) \cdot 10^s$, što je jednako n_0 na osnovu pretpostavke. Pošto je $r' = 10r + n \bmod 10$, onda važi da r' ima jednu cifru više nego r tj. ima $i - s + 1 = i' - s$ cifara), a vodeća cifra je različita od nule (vodeća cifra se nije promenila).

Teorema: Nakon izvršavanja koda važi da je $r = \text{rev}(n_0)$.

Pošto je $n \geq 0$ i nije $n > 0$ (jer se petlja završila), važi da je $n = 0$. Zato je $n_0 = \text{rev}(r) \cdot 10^s$. tj. $\text{rev}(r) = \frac{n_0}{10^s}$. Pošto je vodeća cifra broja r različita od nule i pošto je poslednja cifra broja $\frac{n_0}{10^s}$ takođe različita od nule (jer n_0 ima tačno s završnih nula koje se deljenjem sa 10^s brišu), važi da je $r = \text{rev}(\frac{n_0}{10^s})$. Međutim, važi i da je $\text{rev}(n_0) = \text{rev}(\frac{n_0}{10^s})$, jer $\text{rev}(n_0)$ ima tačno s vodećih nula koje se mogu obrisati bez uticaja na konačni rezultat.

Zaustavljanje se dokazuje prilično jednostavno jer je $n \geq 0$ i konstantno se smanjuje, pa mora doći do nule.

Holandska 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. Dokaži njegovu korektnost.

Ovaj problem, relevantan za particionisanje u okviru algoritma QuickSort u slučaju nizova u kojima se elementi dosta ponavljaju, i njegovo efikasno rešenje uveo je čuveni holandski informatičar Edsger Dejkstra. U originalnoj formulaciji se pretpostavljalo da su elementi niza tri boje holandske trobojke, koje je trebalo sortirati.

Pokušajmo da formalnim rezonovanjem izvedemo algoritam za rešavanje ovog problema. Želimo da postignemo da su u svakom koraku izvršavanja petlje u nizu elementi složeni tako da idu prvo negativni, pa zatim nule, pa zatim nepoznati i na kraju pozitivni (želimo raspored oblika `<<<<000????>>>>`). Granice između ovih segmenata ćemo obeležiti korišćenjem promenljivih. Neka granicu između negativnih i nula označava promenljiva `N`, granicu između nula i nepoznatih označava promenljiva `i`, a granicu između nepoznatih i pozitivnih promenljiva `P`. Pošto promenljiva uvek ukazuje na neki element niza (ne može da pokazuje između elemenata), moramo da se odlučimo na koju stranu granice ćemo postaviti vrednosti tih promenljivih. Pretpostavimo da `N` pokazuje neposredno iza poslednjeg negativnog elementa, `i` neposredno iza poslednje nule, a `P` neposredno iza poslednjeg nepoznatog elementa, tj. da su u intervalu $[0, N)$ svi brojevi negativni, da su u intervalu $[N, i)$ sve nule, da su u intervalu $[i, P)$ svi nepoznati brojevi, a da su u intervalu $[P, n)$ svi brojevi pozitivni. Na osnovu ovoga odredimo inicijalne vrednosti promenljivih. Na početku programa su svi brojevi nepoznati. Odatle sledi da promenljiva `i` mora da dobije vrednost 0, a da promenljiva `P` mora da dobije vrednost n. Jedini način da uslov koji namećemo bude ispunjen je da još promenljiva `N` dobije vrednost 0. Dakle, ako želimo da važi nametnuti raspored, program mora da počne ovako.

```
int i = 0, N = 0, P = n;
```

Program će se izvršavati sve dok ima nepoznatih elemenata. Pošto su oni u intervalu $[i, P)$, program će se izvršavati dok je taj interval neprazan, tj. dok je $i < P$ (kada je $i = P$ interval je prazan). Dakle, dolazimo do programa sledećeg oblika.

```
int i = 0, N = 0, P = n;
while (i < P) {
   ...
}
```

Cilj nam je da napredujemo, tj. da u svakom koraku proširimo invarijantu. Promenljiva `i` ukazuje na prvi nepoznati element. Jasno je da program treba da izvrši analizu njegovog statusa.

```
int i = 0, N = 0, P = n;
while (i < P) {
```

```
    if (a[i] > 0)
      ...
    else if (a[i] < 0)
      ...
    else
      ...
}
```

Ostaje da na osnovu invarijante odredimo šta se u svakom uslovu mora desiti.

- Ako je $a_i > 0$ taj element mora da se pomeri na kraj niza (tamo slažemo pozitivne elemente). Na osnovu invarijante znamo da se prvi poznat pozitivan element (ako postoji) nalazi na poziciji P. Dakle, želimo da a_i dovedemo na poziciju ispred P i on će tada postati prvi pozitivan element. Zato je potrebno da smanjimo P za 1. Na poziciju i dovodimo element sa pozicije $P-1$ koji je nepoznat, tako da u tom slučaju ne treba da diramo ni i, ni N. Dakle, u ovom slučaju treba da umanjimo promenljivu `P` za 1 i onda da razmenimo sadržaj elemenata niza na pozicijama `i` i `P`.

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

- Ako je $a_i < 0$ taj element treba da se pomeri ka početku niza. Znamo da se prva poznata nula (ako postoji) nalazi na poziciji N. Ako bismo razmenili element na mestu i i na mestu N tada bi se niz negativnih elemenata produžio za 1, isto kao i niz nula. Zato bi nakon te razmene trebalo i promenljive `N` i `i` uvećati za 1.

  ```
  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
      ...
  }
  ```

- Na kraju, ako je $a_i = 0$, on samo produžava niz nula koje mu prethode, tako da je jasno da se samo promenljiva `i` može povećati za 1.

  ```
  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++;
  }
```

Primetimo da se tokom izvršavanja algoritma čuvaju dve pozicije koje se tokom izvršavanja algoritma pomeraju stalno u istom smeru. Za takve algoritme kažemo da koriste *tehniku dva pokazivača*.

Ovim smo videli pravu moć formalnog rezonovanja o kodu - na osnovu samo nekoliko polaznih pretpostavki uspeli smo da izvedemo naredbe našeg programa. Ova tehnika je jako tesno povezana sa rezonovanjem o korektnosti programa, a u nastavku ćemo je često koristiti u ovom obliku. Čak i kada ne budemo formalno zapisivali dokaz korektnosti, prilikom programiranja ćemo imati u vidu invarijante koje će nam pomagati da odredimo naredbe od kojih se program mora sastojati, da bi se te invarijante održale.

Na osnovu opisane konstrukcije veoma jednostavno dobijamo i formalan dokaz korektnosti izvedenog algoritma. Ilustracije radi, zapisaćemo ga u celosti, mada on predstavlja samo reformulaciju zaključaka korišćenih tokom izvođenja algoritama.

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 $0 \leq N \leq i \leq P \leq n$.

- 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 (na osnovu invarijante znamo da to važi za intervalu $[0, N)$ koji se nije promenio, a nakon razmene to važi i za a_N). Takođe, možemo da tvrdimo da su u intervalu $[N', i')$ sve nule. Zaista, na osnovu

invarijante to znamo za za interval $[N, i)$, pa i za njegov podinterval $[N', i)$, a pošto je nakon razmene $a_i = 0$, možemo tvrditi da su svi elementi u intervalu $[N', i')$ 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 (jer na osnovu uslova petlje znamo da je $i < P$).

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 $P' = P - 1$. Primetimo da 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.

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.

Binarna pretraga 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). Razmotriti rekurzivnu i iterativnu implementaciju.

Rešenje može biti realizovano narednom varijantom binarne pretrage.

```
int prviParan(int a[], int l, int d) {
    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);
}
```

```
int prviParan(int a[], int n) {
   return prviParan(a, 0, n-1);
}
```

Indukcijom dokazujemo sledeću lemu.

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.

- Baza indukcije je slučaj $l > d$. Na osnovu naše 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 upravo tražena vrednost.
- Pretpostavimo sada da rekurzivni pozivi vraćaju ispravne pozicije i dokažimo da poziv `prviParan(a, l, d)` za vraća ispravnu poziciju i kada nije $l > d$, tj. kada se vrše rekurzivni pozivi. Tada je $0 \leq l \leq d$. Neka je $s = l + \lfloor \frac{d-l}{2} \rfloor$. 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 `prviParan(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.

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).

Pošto je rekurzija repna, ona se veoma jednostavno eliminiše.

```
int prviParan(int a[], int n) {
  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;
}
```

Ovaj program je ekvivalentan prethodnom, rekurzivnom, pa je takođe korektan. Vežbe radi, tu korektnost možemo dokazati i direktno, razmatranjem sledeće invarijante petlje.

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.

- Invarijanta važi pre petlje (zaista iz $l = 0$, $d = n-1$ i $n \geq 0$ sledi da je $0 \leq l \leq d+1$, nijedna nenegativna pozicija nije strogo manja od l i nijedna pozicija strogo veća od d nije strogo manja od n).
- Pretpostavimo da invarijanta važi pre ulaska u petlju i dokažimo da važi i nakon izvršavanja tela petlje. Neka je $s = l + \lfloor \frac{d-l}{2} \rfloor$. 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 $s-1$ (označimo sa $l' = l$ novu vrednost promenljive l, a sa $d' = s-1$ novu vrednost promenljive d) 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 $s+1$ (označimo ponovo $l' = s+1$ i $d' = d$ nove vrednosti promenljivih) invarijanta ostaje održana jer su zaista svi elementi na nenegativnim pozicijama strogo manjim od $l' = s+1$ neparni. Dakle, navedeni uslov jeste invarijanta petlje.

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.

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. Zaista, 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.

Kada je kôd korektan, dokaz je obično neinformativan. Pomaže nam da "mirno spavamo", ali ništa više od toga. Mnogo interesantnija situacija se dešava u slučaju kada nam formalno rezonovanje o kodu pomaže da detektujemo i ispravimo greške u programu (tzv. bagove). Pogledajmo naredni pokušaj implementacije prethodne funkcije.

```
int prviParan(int a[], int n)
  int l = 0, d = n;
  while (l < d) {
```

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

Na osnovu incijalizacije deluje da pokušavamo da pretražimo poluzatvoreni interval $[l, d)$. Pošto je u pitanju binarna pretraga, izgleda da se nameće invarijanta da su svi elementi iz $[0, l)$ neparni, a da su svi iz intervala $[d, n)$ parni. Na početku su oba ta intervala prazna, pa invarijanta za sada dobro funkcioniše. Ako pogledamo uslov petlje, deluje da petlja radi dok se interval nepoznatih elemenata $[l, d)$ ne isprazni (zaista, kada je $l \geq d$, taj interval je prazan). Za sada sve radi kako treba. Pokušamo sada da proverimo da li izvršavanje tela petlje održava invarijantu.

Ako je a_s paran element, tada se promenljiva `d` postavlja na vrednost $d' = s-1$. Na osnovu invarijante treba da važi da su svi elementi u intervalu $[d', n)$ parni. Međutim, mi to ne znamo, jer samo znamo da je a_s paran, ali ne znamo da je a_{s-1} paran. Dakle, ovde se sigurno krije greška u kodu. Ako dodelu `d = s-1` zamenimo sa `d = s`, tada će invarijanta biti održana (jer znamo da je paran a_s, pa će biti parni i svi elementi iza njega).

Ako je a_s neparan, tada se promenljiva `l` postavlja na vrednost $l' = s + 1$. Na osnovu invarijante treba da važi da su svi elementi u intervalu $[0, l')$ neparni, međutim, to će ovde biti ispunjeno, jer je a_s neparno, pa su neparni i svi elementi ispred njega. Dakle, u ovom slučaju je kôd korektan i invarijanta ostaje održana.

Na kraju, kada se petlja završi možemo zaključiti da važi da je $l = d$ (jer sve vreme važi da je $l \leq d$, a nakon petlje ne važi da je $l < d$). U kodu se za poziciju prvog parnog elementa proglašava pozicija $d + 1$. Iako je u originalnoj varijanti koda `l` moglo bez problema da se zameni sa `d+1`, u ovoj varijanti to nije moguće. Naime, mi na osnovu invarijante ovog koda znamo da se na poziciji $l = d$ nalazi paran element, a da se na poziciji $l-1$ nalazi neparan (osim kada je $l = 0$ i tada neparnih elemenata nema). Zato povratna vrednost nije korektna i potrebno je zameniti je sa `return d`, jer se prvi parni element nalazi na poziciji d (osim kada su svi neparni, kada je $d = n$, no i tada je d ispravna povratna vrednost). Dakle, otkrili smo dve greške i ispravljeni kôd u ovoj varijanti treba da izgleda ovako.

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

```
      l = s+1;
  }
  return d;
}
```

Programeri često program ispravljaju tako što nasumice pokušavaju da pomere indekse za 1 levo ili desno, da zamene manje sa manje ili jednako i slično. Već na ovako kratkim programima se vidi da je prostor mogućih kombinacija veliki, a da je mogućnost za grešku prilikom takvog eksperimentalnog pristupa veoma velika. Stoga je uvek bolje zastati, formalno analizirati šta je potrebno da kôd radi i ispraviti ga na osnovu rezultata formalne analize.

Na kraju, skrenimo pažnju na još jedan detalj ispravljenog programa. Parcijalna korektnost je jasna na osnovu analize koju smo sproveli, međutim, zaustavljanje može biti dovedeno u pitanje, s obzirom na naredbu `d = s`. Zaustavljanje dokazujemo tako što pokazujemo da se u svakom koraju smanjuje broj nepoznatih elemenata, tj. da dužina intervala $[l, d)$ koja je jednaka $d - l$ u svakom koraku petlje opada. Postavlja se pitanje da li je to stvarno moguće i u izmenjenom kodu u kome se javlja naredba `d=s`. Odgovor je potrvrdan, a obrazloženje je suptilno. Prvo, na osnovu uslova petlje važi da je $l < d$. Dalje, vrednost s se izračunava naredbom `s = l + (d - l) / 2` što nam na osnovu semantike jezika C++ daje da je $s = \lfloor \frac{l+d}{2} \rfloor$. Zbog zaokruživanja naniže, važi da je $s < d$ i zato se nakon određivanja $d' = s$, $l' = l$ vrednost $d' - l'$ smanjuje u odnosu na $d - l$. Važi i da je $l \leq s$, ali pošto je u drugoj grani $l' = s + 1$ i $d' = d$, vrednost $d' - l'$ se opet smanjuje u odnosu na $d - l$. Pošto je $l \leq d$ invarijanta, smanjivanje ne može trajati doveka, pa se u nekom trenutku program zaustavlja. Da je zaokruživanje kojim slučajem vršeno naviše (npr. `s = l + (d - l + 1) / 2`), program bi mogao upasti u beskonačnu petlju.

Particionisanje u algoritmu QuickSort

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 $0 \leq l \leq d < n$, dokaži njegove invarijante i korektnost.

Prvu varijantu algoritma ćemo organizovati tako da invarijanta bude da se u prvoj fazi algoritma na početku niza nalazi pivot, zatim elementi niza koji su manji ili jednaki pivotu, zatim elementi koji su veći od pivota i na kraju elementi koji još nisu ispitani. Pamtićemo poziciju m koja će ukazivati na granicu između elemenata manjih i jednakih od pivota i onih koji su veći od njega, kao i poziciju i koja će ukazivati na prvi neobrađen (neispitan) element. Želimo da invarijanta petlje bude to da se pivot nalazi na poziciji l, da su na pozicijama (l, m) elementi koji su manji ili jednaki od pivota i da su na pozicijama $[m, i)$ elementi koji su veći od pivota, pri čemu je $l < m \leq i \leq d$.

Da bi invarijanta bila ispunjena na početku, potrebno je da postavimo vrednosti promenljivih tako da važi $m = i = l + 1$, jer su tada oba intervala prazna. U svakom koraku petlje ćemo obrađivati jedan po jedan nepoznati element (i to prvi, onaj na poziciji a_i), dok god ih ima. Pošto su oni u intervalu $[i, d]$, petlju ćemo izvršavati dok god je $i \leq d$. Postoje dve mogućnosti za element a_i.

Ako je a_i veći od pivota, potrebno je samo da povećamo i za jedan da bi invarijanta bila održana. Naime, na osnovu pretpostavke pre izvršavanja tela petlje elementi u intervalu $[m, i)$ su veći od pivota, i pošto je a_i veći od pivota, ako postavimo $i' = i + 1$, veći od pivota će biti svi u intervalu $[m, i')$. Pošto je $m' = m$ i $m \leq i$, svi u intervalu (l, m) će ostati manji ili jednaki pivotu.

Ako je a_i manji ili jednak pivotu možemo ga zamenti sa elementom na poziciji m, i uvećati promenljive `m` i `i` za jedan. Pošto su na osnovu induktivne pretpostavke svi elementi na pozicijama (l, m) bili manji ili jednaki od pivota i pošto je takav element a_i koji nakon razmene dolazi na poziciju m, znamo da će takvi biti svi elementi na pozicijama (l, m'), ako je $m' = m + 1$. Svi elementi na pozicijama $[m, i)$ su bili veći od pivota. U slučaju da je $i = m$ takvih elemenata nema, element a_i će biti razmenjen sam sa sobom (niz će biti nepromenjen) i pošto je $i' = i + 1$ i $m' = m + 1$ interval $[m', i')$ će takođe biti prazan, pa će svi njegovi elementi (kojih nema) biti veći od pivota. Ako je $i > m$, tada se element sa pozicije m za koji na osnovu pretpostavke znamo da je veći od pivota menja sa elementom na poziciji a_i. Elementi sa pozicija $(m+1, i)$ su takođe veći od pivota i pošto je sada takav i element na poziciji i takvi su svi elementi na pozicijama $(m+1, i+1) = (m', i')$. Dakle, i u ovom slučaju telo petlje održava invarijantu.

Na kraju petlje je $i = d + 1$, pa su svi elementi u intervalu (l, m) manji ili jednaki pivotu, a svi u intervalu $[m, n]$ veći od njega. Razmenom elemenata na pozicijama l i $m - 1$ dolazimo u situaciju u kojoj se na pozicijama $[l, m - 1)$ nalaze elementi koji su manji ili jednaki od pivota, pivot se nalazi na poziciji $m - 1$, a elementi na pozicijama $[m, d]$ su veći od pivota. Ako nakon završetka petlje važi $m = l + 1$, ne postoje elementi manji ili jednaki pivotu, razmena elemenata na pozicijama l i $m - 1$, neće proizvesti nikakav efekat, ali će sva navedena tvrđenja i dalje biti tačna.

Programski kôd lako sledi iz prethodne analize (praktično dokaza njegove korektnosti).

```
int m = l+1;
for (int i = l+1; i <= d; i++)
   if (a[i] <= a[l])
     swap(a[i], a[m++])
swap(a[l], a[m-1]);
```

Invarijantu možemo organizovati i 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 $l < m$, $v \leq d$ i $m \leq d + 1$. Na početku možemo inicijalizovati promenljive tako da je

$m = l + 1$ i $v = d$ 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 $m \leq v$. 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 (ovo se neće moći dogoditi u slučaju kada je $m = v$, jer je tada element na toj poziciji ili manji ili jednak ili veći od pivota, pa jedan od prva dva slučaja mora da važi). Kada se petlja završi, poslednji element koji je manji ili jednak pivotu se nalazi na poziciji $m - 1$, pa ga kao i u prethodnoj varijanti razmenjujemo sa pivotom.

Programski kôd lako sledi iz prethodne analize (praktično dokaza njegove korektnosti).

```
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]);
```

Primetimo da ovaj kod možemo organizovati i malo drugačije. Naime, prva dva slučaja možemo iscrpno izvršavati tj. pomerati levi pokazivač sve dok ne dođemo do elementa koji je veći od pivota i desni pokazivač sve dok ne dođemo do elementa koji je manji ili jednak od pivota i onda ih možemo razmeniti.

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

Primetimo da na taj način imamo malo više provera, jer se uslov $m \leq v$ proverava i u unutrašnjim petljama.

Najmanji ne-zbir podskupa

Problem: Dat je skup prirodnih brojeva (zadat u obliku sortiranog niza). Definisati funkciju koja određuje najmanji prirodan broj koji se ne može dobiti kao zbir nekog podskupa elemenata tog skupa i dokazati njenu korektnost.

Činjenica da su elementi sortirani olakšava rešenje zadatka. Obrađivaćemo element po element i održavaćemo granicu do koje smo sigurni da se svaki broj može predstaviti kao zbir nekog podskupa. Dokažimo korektnost narednog algoritma.

```
int najmanjiKojiNijeZbirPodskupa(int a[], int n) {
   int mozeDo = 0;
   for (int i = 0; i < n; i++) {
      if (a[i] > mozeDo + 1)
         return mozeDo + 1;
      mozeDo += a[i];
   }
   return mozeDo + 1;
}
```

Lema: Neka je m vrednost promenljive `mozeDo`. Invarijanta petlje je da je $0 \leq i \leq n$, da je m zbir svih elemenata određenim pozicijama iz intervala $[0, i)$ i da se svaki broj iz intervala $[0, m]$ može dobiti kao zbir nekog podskupa elemenata na pozicijama iz intervala $[0, i)$.

Pre ulaska u petlju je $i = 0$ i $m = 0$, 0 se može dobiti kao zbir praznog podskupa, a interval $[0, i)$ je prazan. Pretpostavimo da tvrđenje važi pre ulaska u petlju.

Ako je $a_i > m + 1$, tvrdimo da je $m + 1$ traženi najmanji broj. Na osnovu invarijante znamo da su svi brojevi iz intervala $[0, m]$ pokriveni, tako da manji broj od $m + 1$ ne može biti rešenje. Dokažimo da broj $m + 1$ ne može biti zbir podskupa. Pošto je niz sortiran, svi elementi od a_i do a_{n-1} su strogo veći od $m + 1$. Dakle, ni jedan od tih elemenata ne sme biti uključen u podskup jer bi njihovim uključivanjem zbir već premašio $m + 1$. Podskup se mora sastojati samo od elemenata a_0 do a_{i-1}, međutim, pošto je m njihov zbir, zbir svakog njihovog podskupa je manji ili jednak m. Dakle, $m + 1$ se ne može postići i on je traženo rešenje.

Ako je $a_i \leq m + 1$, tada je $m' = m + a_i$, $i' = i + 1$ i tvrdimo da je m' zbir svih elemenata a_0, ..., a_i i da se svaki broj iz intervala $[0, m']$ može predstaviti kao zbir nekog podskupa elemenata iz intervala $[0, i') = [0, i]$. Prva tvrdnja je prilično očigledna, jer je po pretpostavci m zbir svih elemenata a_0, ..., a_{i-1}, a $m' = m + a_i$. Na osnovu pretpostavke znamo da svi brojevi iz $[0, m]$ mogu biti zbirovi podskupova elemenata sa pozicija iz intervala $[0, i)$. Slično i svi brojevi iz intervala $[a_i, a_i + m]$ se mogu dobiti kao zbir nekog podskupa elementa iz intervala $[0, i') = [0, i]$. Naime, taj podskup će biti unija elementa a_i i onog podskupa elemenata sa pozicija $[0, i)$ čiji je zbir jednak razlici između tog broja i broja a_i – on je iz $[0, m]$, pa na osnovu pretpostavke takav podskup postoji.

Pošto je $a_i \leq m+1$ unija intervala $[0, m]$ i $[a_i, a_i+m]$ je $[0, a_i+m] = [0, m']$. Zato je svaki element iz $[0, m']$ jednak zbiru nekog podskupa elemenata sa pozicija iz intervala $[0, i'), pa invarijanta ostaje očuvana.

Teorema: Slučaja vraćanja vrednosti funkcije unutar petlje je obrađen. Kada se petlja završi, važi da je $i = n$. Na osnovu invarijante m je zbir svih elemenata niza, i svaki broj iz $[0, m]$ jeste zbir nekog podskupa elemenata niza. Zato je $m+1$ najmanji element koji nije moguće dobiti (jer se uključivanjem svih elemenata dobija najviše m) i funkcija vraća ispravno rešenje.

Sortiranje umetanjem

Problem: Definisati funkciju koja sortira niz korišćenjem algoritma sortiranja umetanjem i dokaži njenu korektnost. Razmotriti iterativnu i rekurzivnu varijantu.

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

void insert(int a[], int k) {
   if (k == 0)
      return;
   if (a[k-1] <= a[k])
      return;
   swap(a[k-1], a[k]);
   insert(a, k-1);
}

void insertionSort(int a[], int k) {
  if (k <= 1)
    return;
  insertionSort(a, k-1);
  insert(a, k-1);
}

int main() {
   int a[] = {3, 5, 8, 4, 2, 6, 1, 9};
   int n = sizeof(a) / sizeof(int);
   insertionSort(a, n);
   for (int i = 0; i < n; i++)
      cout << a[i] << endl;
   return 0;
}
```

Dokažimo prvo indukcijom sledeću lemu o funkciji `insert`.

Lema: Ako je niz a dužine veće ili jednake k, funkcija `insert` korektno umeće element a_k u sortirani prefiks niza a_0, ..., a_{k-1}, tako da nakon umetanja niz a_0,

…, a_k postaje sortiran, pri čemu mu se njegov multiskup elemenata ne menja (on predstavlja permutaciju odgovarajućeg prefiksa početnog niza), kao i da ne menja deo niza iza pozicije k.

- Baza indukcije je slučaj $k = 0$. Funkcija tada ne menja niz, a pošto je svaki jednočlani niz sortiran, tvrđenje važi.
- Pretpostavimo da je prefiks a_0, …, a_{k-1} sortiran i da je $k > 0$. Označimo sa a' rezultujuće stanje niza. Ako je $a_{k-1} \leq a_k$, ceo niz a_0, …, a_k je već sortiran, a funkcija ne menja niz (samim tim ne menja ni elemente iza k) tj. važi $a' = a$, pa tvrđenje važi. U suprotnom je a_{k-1} najveći element niza a_0, …, a_k i on se razmenom sa a_k dovodi na poslednje mesto u nizu tj. važi da je $a'_k = a_{k-1}$ i $a'_{k-1} = a_k$ (ova razmena čuva multiskup elemenata). Pošto je i prefiks a_0, …, a_{k-2} sortiran, na osnovu induktivne hipoteze rekurzivni poziv `insert(a, k-1)` sortira deo niza na pozicijama 0, …, $k-1$, čuva njegov multiskup elemenata i ne menja elemente od pozicije k nadalje. Dakle, niz a'_0, …, a'_{k-1} postaje sortiran i multiskup elemenata mu je jednak multiskupu elemenata niza $a_0, \ldots, a_{k-2}, a_k$. Pošto je $a'_k = a_{k-1}$ neizmenjeno tokom rekurzivnog poziva i pošto je $a_{k-1} > a_k$, važi da je sortiran i ceo niz $a'_0, \ldots, a'_k$ i da je njegov multiskup elemenata jednak multiskupu $a_0, \ldots, a_k$. Deo $a_{k+1}, \ldots, a_{n-1}$ je nepromenjen (jer ga razmena nije menjala, jer na osnovu induktivne hipoteze znamo da rekurzivni poziv nije menjao deo od pozicije k nadalje, a osim razmene i rekurzivnog poziva funkcija nije vršila druge operacije).

Dokažimo sada indukcijom sledeću teoremu o funkciji `insertionSort`.

Teorema: Ako je niz a dužine veće ili jednake od k, poziv `insertionSort(a, k)` sortira elemente a_0, …, a_{k-1}, a elemente od pozicije k nadalje ne menja.

- Ako je $k \leq 1$, funkcija ne vrši nikakve izmene pa se multiskup elemenata ne menja. Deo niza od pozicije 0 do $k-1$ je ili prazan ili jednočlan, pa je trivijalno sortiran.
- Ako je $k > 1$, tada na osnovu induktivne hipoteze znamo da je nakon rekurzivnog poziva `insertionSort(a, k-1)` deo niza a'_0, …, a'_{k-2} sortiran, a deo od pozicije $k-1$ nadalje nepromenjen. Nakon toga poziva se `insert(a, k-1)`. Pošto je njen preduslov ispunjen (jer je prefiks sortiran) na osnovu prethodno dokazane karakterizacije funkcije `insert` znamo da će nakon njenog poziva i niz a'_0, …, a'_{k-1} biti sortiran i da mu se multiskup elemenata neće promeniti i da se deo od pozicije k nadalje neće promeniti.

Pošto u glavnoj funkciji pozivamo `insertionSort(a, n)` gde je n dužina niza, znaćemo da je sortiran deo niza na pozicijama 0 do $n-1$, što je zapravo ceo niz.

Naravno, implementaciju možemo napraviti i iterativno (oslobađanjem rekurzije).

```
void insertionSort(int a[], int n) {
```

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

Prethodnu implementaciju je moguće dodatno poboljšati, ako ciklično pomeranje dela niza umesto razmenama elemenata realizujemo pomoću dodela.

```
void insertionSort(int a[], int n) {
  for (int j = 1; j < n; j++) {
    int tmp = a[j];
    int i;
    for (i = j - 1; i >= 0 && a[i] > tmp; i--)
      a[i+1] = a[i];
    a[i+1] = tmp;
  }
}
```

Da bismo dokazali korektnost ove funkcije, potrebno je prvo da izanaliziramo untrašnju petlju. Želimo da dokažemo da telo unutrašnje petlje ispravno umeće element a_j na njegovo mesto u sortiranom prefiksu a_0, ..., a_{j-1}, ne menjajući pri tom elemente a_{j+1}, ..., a_{n-1}. To dokazujemo razmatranjem invarijante unutrašnje petlje, koja je malo komplikovanija.

Lema: Ako je a stanje niza pri ulasku u telo spoljašnje petlje, a sa a' njegovo tekuće stanje pri izvršavanju unutrašnje petlje, važi:

- $-1 \leq i \leq j-1$
- promenljiva tmp sadrži vrednost a_j,
- deo niza na pozicijama 0, ..., i je nepromenjen u odnosu na stanje koje je važilo pri ulasku u telo spoljne petlje (tj. $a'_0 = a_0$, ..., $a'_i = a_i$),
- deo niza na pozicijama $j+1$, ..., $n-1$ je nepromenjen u odnosu na to stanje ($a'_{j+1} = a_{j+1}$, ..., $a'_{n-1} = a_{n-1}$),
- elementi a'_{i+2}, ..., a'_j svi svi strogo veći od tmp i redom su jednaki elementima a_{i+1}, ..., a_{j-1} niza u stanju koje je važilo pri ulasku u telo spoljne petlje.

Dokažimo ovo.

- Pre ulaska u unutrašnju petlju važi da je $i = j-1$ i invarijanta je trivijalno ispunjena (jer je izvršena dodela `tmp = a[j]`, $a' = a$ jer se niz još nije menjao, pa ni elementi od a_0 do a_{j-1}, kao ni od a_{j+1} do a_{n-1}, a deo niza a_{i+2}, ..., a_j je zapravo prazan jer je $i+2 = j+1$).
- Ako invarijanta važi na ulasku u telo unutrašnje petlje, dokazujemo da će ona važiti i izvršavanja tela (i koraka) te petlje (uz pretpostavku da je i uslov petlje ispunjen). Pokažimo ovo i formalno. Neka je $i' = i-1$ nova vrednost promenljive i (promenljive j i tmp se ne menjaju tokom izvršavanja unutrašnje petlje). Iz pretpostavke da invarijanta važi za neko

i, potrebno je da dokažemo da će ona nakon izvršavanja tela petlje važiti i za $i' = i - 1$.

Pošto pretpostavljamo da invarijanta važi na ulasku u telo unutrašnje petlje, a pošto se u telu unutrašnje petlje ne menja vrednost tmp, znamo da je $tmp = a_j$. Na osnovu pretpostavke znamo da je $a'_0 = a_0$, ..., $a'_i = a_i$. Pošto se u telu unutrašnje petlje menja samo element na poziciji $i + 1$ i pošto je $i' + 1 = i$, znamo da važi da je $a'_0 = a_0$, ..., $a'_{i'} = a_{i'}$.

Takođe, potpuno direktno iz te pretpostavke zaključujemo i da je $a'_{j+1} = a_{j+1}$, ..., $a'_{n-1} = a_{n-1}$, jer se taj deo niza ne menja ni u telu unutrašnje petlje. Na kraju treba još samo da dokažemo da su elementi $a'_{i'+2}$, ..., a'_j strogo veći od tmp i da su jednaki redom elementima $a_{i'+1}$, ..., a_{j-1}. Na osnovu pretpostavke to sigurno važi za elemente a'_{i+2}, ..., a'_j tj. pošto je $i' = i - 1$ za elemete $a'_{i'+3}$, ..., a'_j. Ostaje još samo da se pokaže da je $a'_{i'+2} = a_{i'+1}$, tj. da je $a'_{i+1} = a_i$ i da je taj element manji od tmp. Međutim, to je jednostavna posledica dodele `a[i+1] = a[i]` unutar tela petlje, kao i toga da je uslov petlje ispunjen tj. da je $a_i < tmp$. Dakle, navedeni uslov jeste invarijanta unutrašnje petlje.

Lema: Invarijanta spoljne petlje je:

- $1 \leq j \leq n$
- deo niza a_0, ..., a_{j-1} je sortiran,
- njegov multiskup elemenata odgovara multiskupu elemenata odgovarajućeg segmenta početnog niza a (tj. neka je njegova permutacija)
- deo niza a_j, ..., a_{n-1} je nepromenjen u odnosu na početno stanje niza.

Dokažimo ovo.

- Na ulasku u petlju je $j = 1$ i tada je ova invarijanta trivijalno ispunjena (jednočlan niz a_0 je sortiran, a niz se nije menjao, pa mu se nije menjao ni multiskup, niti deo od pozicije 1 do kraja).

- Na osnovu invarijante unutrašnje petlje i uslova njenog završetka možemo garantovati da će nakon izvršavanja tela i koraka spoljne petlje njena invarijanta biti održana.

  Pretpostavimo da je invarijanta spoljne petlje tačna pri ulasku u telo spoljne petlje i dokažimo da će ona ostati tačna i nakon izvršavanja tela i koraka spoljne petlje. Neka je $j' = j + 1$, neka a označava početno stanje niza, a a' stanje tokom izvršavanja programa. Nakon završetka unutrašnje petlje važi ili da je $i < 0$ ili da je $a'_i \leq tmp$.

  U prvom slučaju, ako je $i < 0$, na osnovu invarijante unutrašnje petlje, znamo da je $i = -1$ i da su svi elementi od a'_1 do a'_j strogo veći od $tmp = a_j$ i da su redom jednaki elementima od a_0 do a_{j-1}. Na osnovu invarijante spoljne petlje znamo da su ti elementi upravo sortiran prefiks početnog niza a. Nakon dodele `a[i+1] = tmp` na poziciju 0 se upisuje vrednost a_j iz originalnog niza tj. važi da je $a'_0 = a_j$. Ta vrednost je strogo

manja od svih elemenata iz dela a'_1 do a'_j, pa je zbog toga taj deo sortiran i predstavlja permutaciju niza a_0, ..., a_j.

U drugom slučaju, ako je $a'_i \leq tmp$, znamo da je deo niza na pozicijama od 0 do i nepromenjen u odnosu na stanje niza pri ulasku u spoljnu petlju, pa je samim tim i sortiran, i pošto je $a'_i \leq tmp$ i pošto je $tmp = a_j$, nakon dodele `a[i+1] = tmp` ćemo znati da je deo a'_0, ..., a'_{i+1} sortiran. Takođe na osnovu invarijante unutrašnje petlje znamo i da su svi elementi od a'_{i+2} do a'_j strogo veći od tmp, pa je zapravo sortiran i ceo niz od a'_0 do a'_j. To da je on permutacija odgovarajućeg prefiksa početnog niza jednostavna je posledica pretpostavke da invarijanta spoljne petlje važi pri ulazu u njeno telo, pa je deo a'_0 do a'_{j-1} permutacija elemenata a_0 do a_{j-1}, da su elementi a_0, ..., a_i ostali na svojim mestima (tj. da je $a'_0 = a_0$, ..., $a'_i = a_i$), da su elementi a_{i+1}, ..., a_{j-1} završili na mestima a'_{i+2}, ..., a'_j (tj. da je $a'_{i+2} = a_{i+1}$, ..., $a'_j = a_{j-1}$), a da je na poziciju a_{i+1} upisan element a_j (tj. $a'_{i+1} = a_j$). Deo niza od pozicije $j+1$ do $n-1$ je nepromenjen (jer na osnovu invarijante znamo da ga ne menja ni unutrašnja petlja, a vidimo i da ga ne menja ni jedna druga naredba u telu spoljne petlje).

Teorema: Ako je n dužina niza a tada poziv `insertionSort(a, n)` ispravno sortirani niz.

Pošto je po završetku spoljne petlje $j = n$, znamo da će nakon njenog završetka niz predstavljati sortiranu permutaciju početnog niza, što je i trebalo dokazati.

Čas 2.1, 2.2, 2.3 - Efikasnost i složenost algoritama

Vrste složenosti i načini njene analize

Pored svojstva ispravnosti programa, veoma je važno i pitanje koliko program zahteva vremena (ili izvršenih instrukcija) 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. Štaviše, potrebno je imati i opšti način za opisivanje i poređenje *efikasnosti* (ili *složenosti*) različitih algoritama. Obično se razmatraju:

- vremenska složenost algoritma;
- prostorna (memorijska) složenost algoritma.

Vremenska i prostorna složenost mogu se razmatrati

- u terminima konkretnog vremena/prostora utrošenog za neku konkretnu ulaznu veličinu;

- u terminima asimptotskog ponašanja vremena/prostora kada veličina ulaza raste.

Vreme izvršavanja programa može biti procenjeno ili izmereno 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. No, vreme izvršavanja programa može biti opisano opštije, u vidu funkcije koja zavisi od ulaznih argumenata.

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 (na slučaju za koji se algoritam najduže izvršava — u analizi vremenske složenosti, ili na slučaju za koji algoritam koristi najviše memorije — u analizi prostorne složenosti). 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.
- *Analiza najboljeg slučaja*, naravno, nikada nema smisla.

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.

U nastavku će, ako nije rečeno drugačije, biti podrazumevana analiza najgoreg slučaja.

Složenost se obično procenjuje na osnovu izvornog koda programa. Savremeni kompilatori izvršavaju različite napredne optimizacije i mašinski kôd koji se izvršava može biti prilično drugačiji od izvornog koda programa (na primer, kompilator može skupu operaciju množenja zameniti efikasnijim bitovskim operacijama, može naredbu koja se više puta izvršava u petlji izmestiti van petlje i slično). Detalji koji se u izvornom kodu ne vide, poput pitanja da li se neki podatak nalazi u keš-memoriji ili je potrebno pristupati RAM-u, takođe mogu veoma značajno da utiču na stvarno vreme izvršavanja programa. Savremeni procesori podržavaju protočnu obradu i paralelno izvršavanje instrukcija, što takođe čini stvarno ponašanje programa drugačijim od klasičnog, sekvencijalnog modela koji se najčešće podrazumeva prilikom analize algoritama. Dakle, stvarno vreme izvršavanja programa zavisi od karakteristika konkretnog računara na kom se program izvršava, ali i od karakteristika programskog prevodioca,

pa i operativnog sistema na kom se program izvršava. Stvarno vreme izvršavanja zavisi i od konstanti sakrivenih u asimptotskim oznakama, međutim, asimptotsko ponašanje obično prilično dobro određuje njegov red veličine (da li su u pitanju mikrosekunde, milisekunde, sekunde, minuti, sati, dani, godine). Ako (pojednostavljeno) pretpostavimo da se svaka instrukcija na računaru izvršava za jednu nanosekundu ($10^{-9}s$), a da broj instrukcija zavisi od veličine ulaza n na osnovu funkcije $f(n)$, tada je vreme potrebno da se algoritam izvrši dat u sledećim tabelama.

Algoritmi čija je složenost odozgo ograničena polinomijalnim funkcijama smatraju se efikasnim.

$n/f(n)$	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3
10	0,003 μs	0,003 μs	0,01 μs	0,033 μs	0,1 μs	1 μs
100	0,007 μs	0,010 μs	0,1 μs	0,644 μs	10 μs	1 ms
1,000	0,010 μs	0,032 μs	1,0 μs	9,966 μs	1 ms	1 s
10,000	0,013 μs	0,1 μs	10 μs	130 μs	100 ms	16,7 min
100,000	0,017 μs	0,316 μs	100 μs	1,67 μs	10 s	11,57 dan
1,000,000	0,020 μs	1 μs	1 ms	19,93 μs	16,7 min	31,7 god
10,000,000	0,023 μs	3,16 μs	0,01 s	0,23 s	1,16 dan	3×10^5 god
100,000,000	0,027 μs	10 μs	0,1 s	2,66 s	115,7 dan	
1,000,000,000	0,030 μs	31,62 μs	1 s	29,9 s	31,7 god	

Algoritmi čija je složenost ograničena odozdo eksponencijalnom ili faktorijelskom funkcijom se smatraju neefikasnim.

$n/f(n)$	2^n	$n!$
10	1 μs	3,63 ms
20	1 ms	77,1 god
30	1 s	$8,4 \times 10^{15}$ god
40	18,3 min	
50	13 dan	
100	4×10^{13} god	

Gornja granica složenosti se obično izražava korišćenjem O-notacije koju ste već izučavali u uvodnim kursevima programiranja. Podsetimo se.

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“.

U nekim slučajevima koristimo i oznaku Θ koja nam ne daje samo gornju granicu, već precizno opisuje asimptotsko ponašanje.

Definicija: Ako postoje pozitivne realne konstante c_1 i c_2 i prirodan broj n_0 takvi da za funkcije f i g nad prirodnim brojevima važi $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ za sve prirodne brojeve n veće od n_0, onda pišemo $f(n) = \Theta(g(n))$ i čitamo „f je veliko 'teta' od g".

Navedimo karakteristike ovih osnovnih klasa složenosti.

- $O(\log n)$ - izuzetno efikasno, npr. binarna pretraga;
- $O(\sqrt{n})$ - "logaritam za one sa jeftinijim ulaznicama" - nemamo najbolja mesta, ali ipak možemo da gledamo utakicu, npr. ispitivanje da li je broj prost, faktorizacija;
- $O(n)$ - optimalno, kada je za rešenje potrebno pogledati ceo ulaz, npr. minimum/maksimum;
- $O(n \log n)$ - "linearni algoritam za one sa jeftinijim ulaznicama", 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.

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. Podsetimo se nekoliko najčešćih rekurentnih jednačina.

U prvoj grupi se problem svodi na problem dimenzije koja je tačno za jedan manja od dimenzije polaznog problema.

- *Jednačina*: $T(n) = T(n-1) + O(1)$, $T(0) = O(1)$. *Primer*: Traženje minimuma niza. *Rešenje*: $O(n)$.
- *Jednačina*: $T(n) = T(n-1) + O(\log n)$, $T(0) = O(1)$. *Primer*: Formiranje balansiranog binarnog drveta. *Rešenje*: $O(n \log n)$.
- *Jednačina*: $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$. *Primer*: Sortiranje selekcijom. *Rešenje*: $O(n^2)$.

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.

- *Jednačina*: $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$. *Primer*: Hanojske kule. *Rešenje*: $O(2^n)$
- *Jednačina*: $T(n) = T(n-1) + T(n-2) + O(1)$, $T(0) = O(1)$. *Primer*: Fibonačijevi brojevi. *Rešenje*: $O(2^n)$

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.

- *Jednačina*: $T(n) = T(n/2) + O(1)$, $T(0) = O(1)$. *Primer*: Binarna pretraga sortiranog niza. *Rešenje*: $O(\log n)$.
- *Jednačina*: $T(n) = T(n/2) + O(n)$, $T(0) = O(1)$. *Primer*: Pronalaženje medijane (središnjeg elementa) niza. *Rešenje*: $O(n)$.
- *Jednačina*: $T(n) = 2T(n/2) + O(1)$, $T(0) = O(1)$. *Primer*: Obilazak potpunog binarnog drveta. *Rešenje*: $O(n)$.
- *Jednačina*: $T(n) = 2T(n/2) + O(n)$, $T(0) = O(1)$. *Primer*: Sortiranje objedinjavanjem. *Rešenje*: $O(n \log n)$.

Ako su granice u samim jednačinama egzaktne, skoro u svim prethodno nabrojanim jednačinama dato rešenje nije samo gornje ograničenje, već je asimptotski egzaktno. Na primer, rešenje jednačine $T(n) = 2T(n/2) + \Theta(n)$, $T(1) = \Theta(1)$ ima rešenje $T(n) = \Theta(n \log n)$. Izuzetak je primer Fibonačijevog niza gde ponašanje jeste eksponencijalno, ali osnova nije 2, već zlatni presek $(1 + \sqrt{5})/2$.

Potpuno formalno i precizno izvođenje i dokazivanje rešenja ovih jednačina neće biti u direktnom fokusu ovog kursa. Mnogo važnije je steći neku intuiciju zašto su rešenja baš takva kakva jesu. Jedan način da se to uradi je da se krene sa "odmotavanjem" rekurzije i da se vidi do čega se dolazi.

Na primer, kod jednačine $T(n) = T(n-1) + O(1)$ i $T(0) = O(1)$, nakon odmotavanja dobijamo da važi $T(n) = T(n-1) + O(1) = T(n-2) + O(1) + O(1) = T(n-3) + O(1) + O(1) + O(1) = ... = T(0) + n \cdot O(1) = O(1) + n \cdot O(1) = O(n)$.

Kod jednačine $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$ slično dobijamo n sabiraka koji su svi $O(n)$ tako da je ukupna suma $O(n^2)$. Postavlja se pitanje da li je ova granica egzaktna tj. da li je moguće da je složenost manja od izvedenog gornjeg ograničenja. Pretpostavimo da je $T(n) = T(n-1) + cn$ i da je $T(0) = O(1)$. Tada je $T(n) = T(n-1) + cn = T(n-2) + c(n-1) + cn = ... = T(0) + c(1 + ... + n) = O(1) + cn(n+1)/2$, tako da je $T(n) = \Theta(n^2)$.

Pokažimo još i šta se dešava sa jednačinom $T(n) = T(n-1) + c \log n$, $T(0) = O(1)$. Odmotavanjem dobijamo $T(n) = T(n-1) + c \log n = T(n-2) + c \log(n-1) + c \log(n) = ... = O(1) + c(\log 1 + ... + \log n)$. Pošto je logaritam rastuća funkcija, svaki od n članova ovog zbira ograničen je odozgo vrednošću $\log n$. Zato je zbir $O(n \log n)$. Dokažimo da ovo ograničenje nije previše grubo. Važi da je $\log 1 + \log 2 + ... + \log n \geq \log(n/2) + \log(n/2 + 1) + ... + \log n$, jer je prvih $n/2$ članova koji su iz sume izostavljeni sigurno nenegativni. Pošto je logaritam rastuća funkcija (za osnovu veću od 1), svi sabirci u ovom zbiru su veći ili jednaki $\log(n/2)$, pa je $\log(n/2) + \log(n/2 + 1) + ... + \log n \geq \log(n/2) + \log(n/2) + ... + \log(n/2) = (n/2) \cdot \log(n/2)$. Stoga je početni zbir logaritama ograničen i odozdo i odozgo funkcijama koje su $\Theta(n \log n)$, pa je i sam $\Theta(n \log n)$. Još jedan način da se ovo pokaže koji se često sreće u literaturi je sledeći. Zbir logaritama, jednak je logaritmu proizvoda, pa zapravo ovde računamo vrednost $\log 1 \cdot ... \cdot n = \log n!$. Po Stirlingovoj formuli $n!$ se ponaša kao $\sqrt{2\pi n}(\frac{n}{e})^n$. Zato se $\log n!$ ponaša kao $n \log n - n + O(\log n)$, pa je ukupan zbir $\Theta(n \log n)$.

Možemo uočiti neke pravilnosti. Kod svih jednačina oblika $T(n) = T(n-1) +$

$f(n)$, $T(0) = c$, nakon odmotavanja dobijamo da je $T(n) = c + f(1) + f(2) + ... + f(n)$, tako da se određivanje asimptotskog ponašanja svodi na sumiranje. Zbir n sabiraka reda $1 + 2 + ... + n$ ima vrednost $n(n+1)/2$, koja je reda $\Theta(n^2)$. To je samo duplo manje od vrednosti zbira $n + ... + n$, koji se sastoji od n sabiraka i ima vrednost n^2. Zbir n sabiraka $\log 1 + ... + \log n$ ima vrednost koja se ponaša kao $n \log n - n$, što je asimptotski isto kao vrednost zbira $\log n + ... + \log n$ koji ima n sabiraka i vrednost $n \log n$. Slično, zbir $1^2 + ... + n^2$ ima vrednost $n(n+1)(2n+1)/6$, što je $\Theta(n^3)$ i samo je oko tri puta manje od zbira $n^2 + ... n^2$ koji ima n sabiraka i vrednost n^3. Iako ovakve generalizacije prete da budu neprecizne, sa malom dozom rezerve se može proceniti da algoritmi u kojima se n puta primenjuje neka operacija složenosti $\Theta(f(k))$ imaju složenost $\Theta(n \cdot f(n))$, čak i kada se operacija u svakom koraku primenjuje nad podacima koji su se za $O(1)$ povećali u odnosu na prethodni korak i samo u krajnjoj instanci imamo $\Theta(n)$ podataka.

Odmotavanjem jednačine $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$ dobijamo $T(n) = 2T(n-1) + O(1) = 2(2T(n-2) + O(1)) + O(1) = 4T(n-2) + 2O(1) + O(1) = 4(2T(n-3) + O(1)) + 2O(1) + O(1) = 8T(n-3) + 4O(1) + 2O(1) + O(1) = ... = 2^n T(0) + (2^{n-1} + ... + 2 + 1) \cdot O(1) = 2^n \cdot O(1) + (2^n - 1) \cdot O(1) = O(2^n)$. Dakle, iako se u svakom rekurzivnom pozivu radi malo posla, rekurzivnih poziva ima eksponencijalno mnogo, što dovodi do izrazito neefikasnog algoritma.

Jednačine zasnovane na dekompoziciji problema na nekoliko manjih potproblema koje su oblika $T(n) = aT(n/b) + O(n^c)$, $T(0) = O(1)$ se rešavaju na osnovu *master teoreme*.

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

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) \text{ , ako je } \log_b a > k \\ \Theta(n^k \log n) \text{ , ako je } \log_b a = k \\ \Theta(n^k) \text{ , ako je } \log_b a < k \end{cases}$$

Nećemo davati dokaz ove teoreme, ali pokušajmo da damo neko intuitivno jasno objašnjenje.

U prvom slučaju se dobija drvo rekurzivnih poziva čiji broj čvorova dominira poslom koji se radi u svakom čvoru. Razmotrimo jednačinu $T(n) = 2 \cdot T(n/2) + O(1)$, $T(1) = O(1)$. Drvo će sadržati $O(n)$ čvorova, a u svakom čvoru će se vršiti posao koji zahteva $O(1)$ operacija. Odmotavanjem rekurentne jednačine, dobijamo $T(n) = 2 \cdot T(n/2) + O(1) = 4 \cdot T(n/4) + 2 \cdot O(1) + O(1) = 8 \cdot T(n/8) + 4 \cdot O(1) + 2 \cdot O(1) + O(1) = 2^k \cdot T(n/2^k) + (2^{k-1} + ... + 2 + 1) \cdot O(1)$. 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 $\Theta(n)$. I kada n nije stepen dvojke, dobija se isto asimptotsko ponašanje (što se može dokazati ograničavanjem odozgo i odozdo stepenima dvojke).

U drugom slučaju su broj čvorova i posao koji se radi na neki način uravnoteženi. Razmotrimo jednačinu $T(n) = 2 \cdot T(n/2) + c \cdot n$, $T(1) = O(1)$ i ponovo pokušajmo da je odmotamo. $T(n) = 2 \cdot T(n/2) + c \cdot n = 2 \cdot (2 \cdot T(n/4) + c \cdot n/2) + c \cdot n = 4T(n/4) + c \cdot n + c \cdot n = 4(2T(n/8) + c \cdot n/4) + 2 \cdot c \cdot n = 8T(n/8) + 3 \cdot c \cdot n = ... = 2^k \cdot T(n/2^k) + k \cdot c \cdot n$. 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 $n \cdot O(1) + \log_2 n \cdot c \cdot n = \Theta(n \log n)$. Isto važi i kada n nije stepen dvojke.

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 $O(1) + cn(2 - 2/n) = \Theta(n)$.

Prokomentarišimo da se u nekim problemima dobijaju jednačine koje nisu baš u svakom rekurzivnom pozivu identične ovim navedenim. Na primer, prilikom analize algoritma QuickSort, ako je pivot tačno na sredini niza, važi da je $T(n) = 2T(n/2) + O(n)$ i $T(1) = O(1)$. Kada bi se to stalno događalo, rešenje bi bilo $T(n) = O(n \log n)$, međutim, verovatnoća da se to dogodi je strašno mala, jer u većini slučajeva pivot ne deli niz na dva dela potpuno iste dimenzije i zato treba biti obazriv. Ako bi se desilo da pivot stalno završavao na jednom kraju niza, jednačina bi bila $T(n) = T(n-1) + O(n)$, $T(1) = O(1)$, što bi dovelo do složenosti $O(n^2)$, što i jeste složenost najgoreg slučaja. Analizom koju ćemo prikazati kasnije se može utvrditi da je prosečna složenost $O(n \log n)$ tj. da iako pivot nije stalno na sredini, da je u dovoljnom procenu slučajeva negde blizu nje (recimo između 25% i 75% dužine niza). Slična analiza važi i za problem pronalaženja medijane. Međutim, postoje i drugačiji slučajevi. Prilikom obilaska binarnog drveta, balansiranost nema uticaja. Naime, ako je drvo potpuno, tada je jednačina $T(n) = 2T(n/2) + O(1)$, $T(1) = O(1)$, čije je rešenje $O(n)$. Međutim, čak i kada je drvo izdegenerisano u listu, jednačina je $T(n) = T(n-1) + O(1)$, $T(1) = O(1)$, čije je rešenje opet $O(n)$. Kakav god da je odnos broja čvorova u levom i desnom poddrvetu rešenje će biti $O(n)$. To se može opisati jednačinom $T(n) = T(k) + T(n-k-1) + O(1)$, $T(1) = O(1)$, za $0 \leq k \leq n-1$, čije će rešenje biti $O(n)$, bez obzira na to kakvo se k pojavljuje u raznim rekurzivnim pozivima.

Primeri analize složenosti najgoreg slučaja

Euklidov algoritam

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

Razmotrimo prvo originalnu varijantu algoritma, u kojoj se koristi oduzimanje.

```
int nzd(int a, int b) {
   if (a > b)
      return nzd(a-b, b);
   else if (b > a)
      return nzd(a, b-a)
   else
      return a;
}
```

Zaustavljanje se obezbeđuje time što pri svakom rekurzivnom pozivu jedan od elemenata opada (jer su oba broja pozitivna i uvek se manji oduzima od većeg). Do kraja stiže u najviše $a + b$ koraka (a često i manje). Najgori slučaj nastupa ako je jedan od dva broja jednak 1, a drugi dosta veći od njega, jer se onda ta jedinica oduzima (npr. ako je $b = 1$, do kraja se stiže u $a - 1$ koraka). Dakle, ukupna složenost je linearna tj. $O(a + b)$.

Razmotrimo sada varijantu zasnovanu na celobrojnom deljenju.

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

Primetimo da se posle najviše jednog koraka osigurava da je $a > b$ (jer se u svakom koraku par (a, b) menja parom $(b, a \bmod b)$, a uvek važi da je $a \bmod b < b$). Posle bilo koje dve iteracije se od para (a, b) dolazi do para $(a \bmod b, b \bmod (a \bmod b))$ (naravno, pod pretpostavkom da je $b \neq 0$ i da je $a \bmod b \neq 0$). Dokažimo da je $a \bmod b < a/2$. Ako je $b \leq a/2$, tada je $a \bmod b < b \leq a/2$. U suprotnom, za $b > a/2$ važi da je $a \bmod b = a - b < a/2$. Zato se prvi argument posle svaka dva koraka smanji bar dvostruko. Do vrednosti 1 prvi argument stigne u logaritamskom broju koraka u odnosu na veći od polazna dva broja i tada drugi broj sigurno dostiže nulu (jer je strogo manji od prvog) i postupak se završava. Dakle, složenost je logaritamska u odnosu na veći od dva broja, što može da se zapiše i kao $O(\log(a + b))$.

Dakle, varijanta zasnovana na celobrojnom deljenju je neuporedivo efikasnija od varijante zasnovana na oduzimanju.

QuickSort particionisanje

Problem: Proceniti složenost narednog algoritma particionisanja niza dužine $n \geq 1$.

```
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]);
```

Jedna od najčešćih zabluda je da se složenost algoritama procenjuje samo na osnovu broja ugnežđenih petlji. Pošto se u ovom zadatku javlja petlja u petlji, neko bi naivno mogao da kaže da je složenost u ovom slučaju $O(n^2)$, međutim to nije tačno. U ovom algoritmu se koriste dva pokazivača m i v koja praktično kreću sa dva kraja niza i približavaju se jedan drugome, sve dok se ne susretnu. Jasno je, dakle, da je najveći broj koraka koji je moguće napraviti ograničen dužinom niza tj. da je ukupna složenost $O(n)$. Ovaj algoritam spada u grupu tzv. algoritama zasnovanih na tehnici *dva pokazivača* (engl. two pointer), koji se prepoznaju po tome što se dva indeksa niza (tzv. pokazivača) pomeraju kroz niz, ali stalno u istom smeru (nijedan pokazivač se ne vraća nikada unazad). 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.

Analiza prosečne složenosti

Analiza prosečne složenosti 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 je jednaka verovatnoća da će proizvoljni element biti izabran 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 takozvana *jednačina sa potpunom istorijom*, jer se vrednost $T(n)$ 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. U ovom slučaju se svaki od

sabiraka $T(i)$ deli sa n, te pre oduzimanja svaki od uzastopnih članova treba pomnožiti sa odgovarajućim faktorom. Tako se dobija

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= \left(n(n-1) + 2\sum_{i=0}^{n-1} T(i) \right) - \left((n-1)(n-2) + 2\sum_{i=0}^{n-2} T(i) \right) \\ &= 2(n-1) + 2T(n-1) \end{aligned}$$

Zato je

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

Iako je ova jednačina linearna rekurentna jednačina koja povezuje samo dva uzastopna člana niza, ona nije sa konstantnim koeficijentima i potrebno je malo inventivnosti da bismo je rešili. Deljenje sa $n+1$ nam daje pogodniji oblik.

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

Naime, sada se vidi da su dva člana koja uključuju nepoznatu funkciju $T(n)$ istog oblika, pa jednačinu možemo jednostavno odmotati i korišćenjem činjenice da je $T(0) = 0$ dobiti

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

Centralno pitanje postaje kako izračunati zbir

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

Tome pomaže razdvajanje sabiraka na parcijalne razlomke. Iz jednačine

$$\frac{i-1}{i(i+1)} = \frac{A}{i} + \frac{B}{i+1}$$

sledi da je $Ai + A + Bi = i - 1$, pa je $A = -1$, $B = 2$ i važi

$$\frac{i-1}{i(i+1)} = \frac{2}{i+1} - \frac{1}{i}$$

Zato je

$$\sum_{i=1}^{n} \frac{i-1}{i(i+1)} = \frac{2}{1+1} - \frac{1}{1} + \frac{2}{2+1} - \frac{1}{2} + \frac{2}{3+1} - \frac{1}{3} + ... + \frac{2}{n+1} - \frac{1}{n}$$

Možemo grupisati razlomke sa istim imeniocem i dobiti

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

Zato je

$$T(n) = 2(n+1) \cdot \left(1 + \frac{1}{2} + ... + \frac{1}{n}\right) - 4n$$

Znamo da se harmonijski zbir $1+1/2+...+1/n$ asimptotski ponaša kao $\log n+\gamma$, gde je γ Ojler-Maskeronijeva konstanta $\gamma \approx 0,57722$ i zato je

$$T(n) = \Theta\left(2(n+1)(\log n + \gamma) - 4n\right) = \Theta(n \log n).$$

Empirijska analiza složenosti

Umesto direktne analize prosečne složenosti, krenimo empirijskim putem – brojeći operacije u narednoj implementaciji algoritma QuickSort.

```
void quicksort(int l, int d) {
    int i, p;
    if (l >= d) return;
    swap(l, randint(l, d));
    p = l;
    for (i = l+1; i <= d; i++)
        if (x[i] < x[l])
            swap(++p, i);
    swap(l, p);
    quicksort(l, p-1);
    quicksort(p+1, u);
}
```

Ukoliko se kao relevantna operacija izabere poređenje (broj razmena nije veći od broja poređenja), ona se mogu izbrojati, uvećavanjem globalne brojačke promenljive `brojPoredjenja` neposredno pre svakog poređenja.

```
    for (i = l+1; i <= d; i++) {
        brojPoredjenja++;
        if (x[i] < x[l])
            swap(++p, i);
    }
```

Lako je uočiti za koliko se u datoj petlji uvećava promenljiva `brojPoredjenja` i izbeći povećavanje u svakom koraku.

```
    brojPoredjenja += d-l;
    for (i = l+1; i <= d; i++)
        if (x[i] < x[l])
            swap(++p, i);
```

Ukoliko je cilj razmatrati broj operacija, a ne stvarno sortirati niz, sadržaj niza nema značaja, već je samo bitno uvećavanje promenljive `brojPoredjenja`. Otud se u svrhe ovakve analize kôd može pojednostaviti (pri čemu se empirijski može utvrditi da promena ne utiče na rezultat koji funkcija izračunava).

```
void quickcount(int l, int d) {
    if (l >= d) return;
    int p = randint(l, d);
    brojPoredjenja += d-l;
    quickcount(l, p-1);
    quickcount(p+1, d);
}
```

Štaviše, ni indeksi l i d nisu od značaja, već samo širina intervala koji obuhvataju.

```
void quickcount(int n) {
    if (n <= 1) return;
    int p = randint(0, n-1);
    brojPoredjenja += n-1;
    quickcount(p);
    quickcount(n-p-1);
}
```

Umesto uvećavanja globalne brojačke promenljive, funkcija može vratiti broj poređenja kao svoju povratnu vrednost.

```
int quickcount(int n) {
    if (n <= 1) return 0;
    int p = randint(0, n-1);
    return n-1 + quickcount(p) + quickcount(n-p-1);
}
```

Data implementacija izračunava broj poređenja pri jednom sortiranju jednog niza dužine n. Kako su sve vrednosti p jednako verovatne, tačno prosečno vreme se može dobiti uprosečavanjem broja operacija po svim mogućim vrednostima promenljive p.

```
double meancount(int n) {
    if (n <= 1) return 0;
    double zbir = 0;
    for (int p = 0; p < n; p++)
        zbir += n-1 + meancount(p) + meancount(n-p-1);
```

```
    return zbir / n;
}
```

Očigledan problem ovog koda je eksponencijalna računska složenost usled koje program nije moguće izvršiti već za relativno male vrednosti broja n. Neefikasnost dolazi iz velikog broja izračunavnaja istih vrednosti.

Stvar se može značajno popraviti ako se primeti da se u petlji `for` funkcija `meancount` poziva redom za vrednosti `0` i `n-1`, zatim `1` i `n-2` itd. sve do vrednosti `n-1` i `0`. Dakle, za svaku vrednost od `0` do `n-1` dva puta se računa vrednost `meancount` i dodaje na zbir. Stoga se prethodni kôd može transformisati na sledeći način.

```
double meancount(int n) {
    if (n <= 1) return 0;
    double zbir = 0;
    for (int p = 0; p < n; p++)
        zbir += n-1 + 2*meancount(p);
    return zbir / n;
}
```

Ovim se dobija mnogo efikasnija implementacija, međutim i u njoj se isti rekurzivni pozivi izvršavaju više puta. Stvar se može popraviti tehnikom dinamičkog programiranja tako što se sve vrednosti pri prvom izračunavanju upisuju u niz iz kog se čitaju kada su potrebne naredni put.

```
double meancount(int N) {
    dp[0] = dp[1] = 0;
    for (int n = 2; n <= N; n++) {
       double zbir = 0;
       for (int p = 0; p < n; p++)
          zbir += n-1 + 2*dp[p];
       dp[n] = zbir / n;
    }
    return dp[N];
}
```

Prethodni program se može malo uprostiti ako se izbegne dodavanje člana `n-1` i množenja sa `2` u svakom koraku petlje (to se može uraditi samo jednom, nakon petlje).

```
double meancount(int N) {
    dp[0] = dp[1] = 0;
    for (int n = 2; n <= N; n++) {
       double zbir = 0;
       for (int p = 0; p < n; p++)
          zbir += dp[p];
       dp[n] = n-1 + 2*zbir/n;
    }
```

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

Unutrašnja petlja se može eliminisati ako se primeti da se zbir može računati inkrementalno.

```
double meancount(int N) {
    dp[0] = dp[1] = 0;
    double zbir = 0;
    for (int n = 2; n <= N; n++) {
        zbir += dp[n-1];
        dp[n] = n-1 + 2*zbir/n;
    }
    return dp[N];
}
```

Pošto se u svakom koraku petlje koristi samo prethodni element niza, ceo niz se može ukloniti i zameniti samo jednom promenljivom.

```
double meancount(int N) {
    double d = 0;
    double zbir = 0;
    for (int n = 2; n <= N; n++) {
        zbir += d;
        d = n-1 + 2*zbir/n;
    }
    return d;
}
```

Vrednosti promenljivih `d` i `zbir` koje program izračunava mogu se direktno opisati rekurentnim jednačinama izvedenim iz koda.

$$T_0 = 0, \quad S_0 = 0$$
$$S_n = S_{n-1} + T_{n-1}$$
$$T_n = n - 1 + 2\frac{S_n}{n}$$

Uvrštavajući T_n u S_n, dobija se

$$S_0 = 0, S_1 = 0$$

i za $n \geq 2$

$$S_n = S_{n-1} + n - 2 + 2\frac{S_{n-1}}{n-1} = \frac{n+1}{n-1}S_{n-1} + n - 2$$

Odade sledi

$$\frac{S_n}{n+1} = \frac{S_{n-1}}{n-1} + \frac{n-2}{n+1}$$

Deljenjem obe strane sa n dobija se

$$\frac{S_n}{(n+1)n} = \frac{S_{n-1}}{n(n-1)} + \frac{n-2}{n(n+1)}$$

Kako su oba člana koja uključuju veličinu S_n iste forme, jednačina se može razmotati i dobija se da za $n \geq 2$ važi

$$\frac{S_n}{(n+1)n} = \sum_{i=2}^{n} \frac{i-2}{i(i+1)} = \sum_{i=2}^{n} \left(\frac{3}{i+1} - \frac{2}{i} \right) = \frac{3}{n+1} + \sum_{i=3}^{n} \frac{1}{i} - \frac{2}{2}$$

tj.

$$S_n = n(n+1) \left(\frac{3}{n+1} + \sum_{i=1}^{n} \frac{1}{n} - \frac{5}{2} \right)$$

Uvrštavanjem ove relacije u T_n dobija se

$$T_n = n - 1 + 2 \left(3 + (n+1) \sum_{i=1}^{n} \frac{1}{n} - \frac{5(n+1)}{2} \right) = 2(n+1) \sum_{i=1}^{n} \frac{1}{i} - 4n$$

Korišćenjem asimptotske ocene harmonijskog reda, dobija se $T(n) = \Theta(n \log n)$.

Ovaj primer pokazuje kako se teorijski rezultat može elegantno izvesti iz empirijskog razmatranja, kao i da granica između programiranja i teorijske analize svojstava algoritma nekada može biti tanka.

Naglasimo i kako je korektnost formula takođe moguće kontrolisati empirijski. Na primer, rekurentna veza za S se može prekontrolisati tako što se proveri da naredna funkcija `T` vraća istu vrednost kao i funkcija `meancount`.

```
double S(int n) {
  if (n <= 1) return 0;
  return (n+1.0)/(n-1.0)*S(n-1)+n-2.0;
}

double T(int n) {
  return n-1 + 2.0*S(n)/n;
}
```

Korektnost zatvorenog oblika za S se može proveriti narednom definicijom.

```
double H(int n) {
  double zbir = 0.0;
  for (int i = 1; i <= n; i++)
     zbir += 1.0/i;
  return zbir;
}

double S(int n) {
  return n*(n+1.0)*(3/(n+1.0) + H(n) - 5.0/2.0);
}

double T(int n) {
  return n-1 + 2.0*S(n)/n;
}
```

Na kraju, može se pokazati i da konačni oblik za T vraća isti rezultat kao i sve prethodne funkcije.

```
double T(int n) {
   return 2*(n+1)*H(n) - 4.0*n;
}
```

Pitanje koje bi matematičar logično mogao postaviti u kontekstu provere izvedenih formula programima je – čemu empirijska provera za nešto što je dokazano? Kao što se greške mogu potkrasti u programima, mogu se potkrasti i u dokazima koje čovek izvodi na papiru. I, stoga, kao što takvo dokazivanje korektnosti programa doprinosi našem uverenju da će raditi korektno, empirijska provera izvedenih formula pomoću programa doprinosi našem uverenju da smo matematička izvođenja uradili korektno. Štaviše, tokom izvođenja reda za $S_n/((n+1)n)$, autori su izostavili član $3/(n+1)$ koji figuriše u izvedenoj formuli. Naime, ovaj član je granični element jedne od suma koje se oduzimaju, a upravo granični slučajevi bivaju najčešće zaboravljeni. Takve greške nije teško uočiti korišćenjem empirijske provere.

Amortizovana analiza složenosti

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 (ako postoji dovoljno izvršavanja te operacije koja će trajati kratko). 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. Ilustrujmo je na jednom primeru.

Jedna od najčešće upotrebljavanih struktura podataka je dinamički niz. Razmotrimo 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 (ništa se suštinski ne bi promenilo u analizi i da je inicijalni kapacitet neki broj m_0). Izbrojmo koliko je puta potrebno izvršiti upis elementa u niz (ta operacija je obično najsporija), 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. Dakle, zbir operacija je onda jednak $k + k + k + 2k + k + 3k + ...$ Da bi se smestilo n elemenata, realokaciju je potrebno vršiti oko n/k puta, pa će ukupan broj operacija biti otprilike jednak

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

Dakle, 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 $O(n)$, doduše, za dosta malu vrednost konstante uz n (što je veće k, to je realokacija manje, pa je cena operacije manja, ali se cena plaća kroz veće zauzeće memorije i manju popunjenost alociranog prostora).

Druga strategija može biti geometrijska i ona podrazumeva da se svaki put veličina niza poveća q puta za neki faktor $q > 1$. Pretpostavimo da je početna veličina niza m_0. Dakle, 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 pritom se prepisuje qm_0 elemenata. Nakon toga se upisuje preostalih $q^2m_0 - qm_0$ elemanata. Postupak se dalje nastavlja po istom principu. Dakle, ukupan broj upisa u niz jednak 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 ukupan broj upisa jednak je

$$m_0(1 + ... + q^r) = m_0\frac{q^{r+1} - 1}{q - 1}.$$

Ako pretpostavimo da je ceo niz popunjen posle r realokacija, tj. da je $n = m_0 q^r$, onda je ukupan broj operacija potrebnih za popunjavanje niza jednak

$$m_0 \frac{q^{\frac{n}{m_0}} - 1}{q - 1} = \frac{qn - m_0}{q - 1}.$$

Asimptotski je, dakle, ukupna cena izvođenja svih operacija jednaka $O(n)$, a amortizovana cena izvođenja jedne operacije dodavanja u ovakav niz je $O(1)$. Konstantan faktor jednak je $\frac{q}{q-1}$ i on je sve manji kako q raste. Zaista, sa povećanjem q vrši se sve manje realokacija, ali se cena plaća većim angažovanjem memorije tj. manjom popunjenošću niza.

Amortizovana analiza složenosti nam pokazuje da sa stanovišta vremena izvršavanja geometrijska strategija realokacije daje značajno bolje rezultate nego aritmetička.

Neki načini poboljšanja složenosti

U nastavku ćemo prikazati kako se neki problemi mogu rešiti različitim algoritmima i analiziraćemo njihovu složenost (slično ste već u kursu P2 radili na primerima algoritama sortiranja i pretrage). Ujedno ćemo videti kako se različite algoritamske tehnike o kojima će detaljno biti reči u nastavku ovog kursa (neke od njih ste već sreli u kursu P2) primenjuju da bi se dobio algoritam bolje složenosti.

Zamena iteracije eksplicitnom formulom

Zbir prvih n prirodnih brojeva

Problem: Definisati efikasan algoritam koji izračunava zbir $1 + 2 + ... + n$. Proceni mu složenost.

Jedno rešenje je da se zadata reši u složenosti $O(n)$, tao što se primeni petlja i algoritam sabiranja serije elemenata.

```
int zbir = 0;
for (int i = 1; i <= n; i++)
   zbir += i;
```

Međutim, 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;
```

Prvi algoritam prvih milijardu prirodnih brojeva sabere za oko $0,438$ sekundi, a drugi za oko $0,004$ sekunde (u to je uključeno i vreme pokretanja programa, ispisa, završetka programa, tj. celokupno vreme izvršavanja prijavljeno pomoću funkcije `time`).

Nedostajući broj

Problem: Dat je niz od n elemenata koji sadrži različite brojeve iz skupa $0, 1, 2, \ldots, n$ (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 $O(n^2)$. Može se primetiti i da nije moguće da se u svakom koraku linearne pretrage dogodi najgori slučaj. Najgori slučaj se dešava kada se element ne nalazi u nizu, a svi elementi koji se traže (sem jednog) su prisutni. Zaista, kada tražimo element koji se nalazi na poziciji i ta pretraga će se završiti u i koraka. Preciznije, element na poziciji 1 će se pronaći u jednom koraku, element na poziciji 2 u dva koraka, element na poziciji 3 u tri koraka i tako dalje. Ukupan broj koraka je zato $1 + 2 + \ldots + n$, što je opet $O(n^2)$.

Bolje rešenje može biti zasnovano na sortiranju elemenata (koje se može uraditi u vremenu $O(n \log n)$) 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 $O(n \log n)$. Kasnije ćemo videti da zahvaljujući specifičnosti raspona brojeva sortiranje možemo uraditi i u vremenu $O(n)$, međutim postoji veoma jednostavan, potpuno matematički postupak koji nam rešenje daje u toj složenosti.

Zbir svih elemenata iz skupa $\{0, 1, 2, \ldots, n\}$ je $0 + 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$. To je zbir elemenata koji se nalaze u nizu i nedostajućeg elementa. Nedostajući element je, dakle, jednak, razlici između $\frac{n(n+1)}{2}$ i zbira svih elemenata niza koji lako možemo izračunati u vremenu $O(n)$.

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

int main() {
  int n;
  int zbir = 0;
  for (int i = 0; i < n; i++) {
     int x;
     cin >> x;
     zbir += x;
  }
  cout << n*(n+1) / 2 - zbir << endl;
```

```
  return 0;
}
```

Posmatrajmo sledeće uopštenje ovog problema.

Problem: Dat je niz od $n-1$ elemenata koji sadrži različite brojeve iz skupa $0, 1, 2, \ldots, n$ (tačno dva broja nedostaju). Odredi koji su brojevi nedostajući.

Problem rešavamo slično kao u prethodnom slučaju, jedino što pored zbira elemenata koristimo i zbir kvadrata. Neka su nedostajući elementi x i y, a neka je zbir postojećih elemenata u nizu z_1, a zbir njihovih kvadrata z_2. Važi da je $z_1 + x + y = 0 + 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$, a da je $z_2 + x^2 + y^2 = 0^2 + 1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$. Zato znamo da je $x + y = \frac{n(n+1)}{2} - z_1 = c_1$, a da je $x^2 + y^2 = \frac{n(n+1)(2n+1)}{6} - z_2 = c_2$, za neke konstante c_1 i c_2. Zato je $(x+y)^2 - (x^2 + y^2) = 2xy = c_1^2 - c_2$. Pošto je $y = c_1 - x$, važi da je $2x(c_1 - x) = c_1^2 - c_2$, pa je $2x^2 - 2c_1 x + c_1^2 - c_2 = 0$. Zato je

$$x = \frac{2c_1 + \sqrt{4c_1^2 - 8(c_1^2 - c_2)}}{4} = \frac{c_1 + \sqrt{2c_2 - c_1^2}}{2},$$

$$y = c_1 - x = \frac{c_1 - \sqrt{2c_2 - c_1^2}}{2}.$$

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

int main() {
   int n;
   cin >> n;
   int z1 = 0, z2 = 0;
   for (int i = 0; i < n-1; i++) {
     int x;
     cin >> x;
     z1 += x; z2 += x*x;
   }
   int c1 = n*(n+1)/2 - z1;
   int c2 = n*(n+1)*(2*n+1)/6 - z2;
   int x = (c1 + sqrt(2*c2 - c1*c1)) / 2;
   int y = c1 - x;
   cout << x << " " << y << endl;
   return 0;
}
```

Za tri nedostajuća elementa mogli bismo upotrebiti i zbir kubova. Moguće je čak uopštenje na k nedostajućih elemenata, ali matematika bi postajala sve teža i teža (ključna tehnika je rešavanje sistema jednačina u kojima figuriše zbir kubova, zbir kvadrata i običan zbir, za koji se može pokazati da se uvek može

eksplicitno rešiti). Složenost bi sve vreme bila jednaka $O(n)$, mada bi konstanta postajala sve veća i veća. Kasnije ćemo konstruisati direktno, algoritamsko rešenje ovog zadatka, bez previše upotrebe matematike.

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. Proceni mu složenost.

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.

```
int broj = 0;
for (int x = a; x <= b; x++)
   if (x % k == 0)
      broj++;

cout << broj << endl;
```

Složenost ovog algoritma, jasno je $O(b - a)$.

Međutim, zadatak možemo rešiti i u složenosti $O(1)$.

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 $n_l = \lceil \frac{a}{k} \rceil$. Najveće n koje zadovoljava drugu nejednačinu jednako je $n_d = \lfloor \frac{b}{k} \rfloor$. 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 $[n_l, n_d]$. Zato je traženi broj brojeva iz intervala $[a, b]$ koji su deljivi brojem k jednak broju 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 tj. ako je $n_d < n_l$.

```
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;
```

Još jedan elegantan način da se ovaj zadatak reši u složenosti $O(1)$ je da se broj deljivih brojeva u intervalu $[a, b]$ predstavi kao razlika između broja deljivih u intervalu $[0, b]$ i broja deljivih u intervalu $[0, a - 1]$ (specijalni slučaj je kada je $a = 0$ i kada samo treba izračunati broj deljivih u intervalu $[0, b]$). Dakle broj elemenata u proizvoljnom segmentu razlažemo na razliku broja elemenata u dva prefiksa). Broj deljivih u bilo kom intervalu oblika $[0, n]$ možemo jednostavno odrediti određivanjem najvećeg broja u tom intervalu deljivog sa k, tj. njegovog količnika sa k. To je broj $\lfloor \frac{n}{k} \rfloor + 1$. Najmanji broj deljiv sa k u svakom takvom prefiksu je 0.

```
broj = a > 0 ? b / k - (a - 1) / k : (b / k) + 1;
```

Za $a = 0$, $b = 10^9$ i $k = 43$ prvi algoritam rezultat vraća za oko $2,602$ sekundi, a drugi i treći za oko $0,004$ sekunde.

Uštedu vremena smo dobili tako što smo u potpunosti eliminisali pretragu.

Eliminisanje identičnih izračunavanja

Jedna od najvažnijih stvari prilikom pravljenja efikasnih algoritama je izbegavanje izračunavanja istih stvari više puta.

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.

Na osnovu definicije niza $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ za $n \geq 2$, rekurzivno rešenje je moguće implementirati veoma jednostavno.

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

Jednačina koja određuje složenost ove funkcije (na primer, u broju sabiranja) 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. Na primer, da bi se izračunao F_5 vrše se rekurzivni pozivi koji izračunavaju F_4 i F_3. Tokom izračunavanja F_4 računa se i F_3, da bi se onda ta izračunata vrednost zanemarila i računala iz početka. Vrednosti F_2, F_1 i F_0 se tokom izračunavanja F_5 izračunavaju više puta.

U slučajevima preklapajućih rekurzivnih poziva, kao što je ovaj, može se primeniti tehnika *dinamičkog programiranja* o kojoj će mnogo više reči biti kasnije. 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*.

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

Već ovom transformacijom složenost je svedena na $O(n)$.

Moguće je u potpunosti eliminisati rekurziju i niz popunjavati redom.

```
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];
}
```

Na kraju, moguće je primetiti i da se u svakom trenutku 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;
}
```

Na osnovu prve definicije vrednost F_{40} se izvršava oko $0,484$ sekunde, a na osnovu svih ostalih za oko $0,005$ sekundi.

U kasnijem bavljenju dinamičkim programiranjem, videćemo da se sličan niz koraka može sprovesti za poboljšanje efikasnosti u mnogim situacijama u kojima se rekurzivni pozivi preklapaju, tj. kada se isti rekurzivni pozivi vrše veći broj puta, čime se narušava efikasnost.

Napomenimo i da se eksplicitnim rešavanjem rekurentne jednačine za Fibionačijeve brojeve dobija Bineova formula koja pokazuje da je $F_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$, gde je $\phi = \frac{1+\sqrt{5}}{2}$, vrednost takozvanog zlatnog preseka.

Najmanja tačna perioda niza

Nekada je čak moguće problem razložiti na manje potprobleme koji su potpuno identični i onda rešiti samo jedan od njih. Razmotrimo sledeći interesantan primer.

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. Na primer, dužina najkraće tačne periode niza `1 2 3 4 1 2 3 4` je 4, dok je za niz `1 2 3 1 2 3 1 2` dužina najkraće tačne periode 8). Definisati funkciju koja efikasno određuje dužinu p najmanje tačne periode tog niza i proceni joj složenost.

Jasno je da dužina periode mora da deli dužinu niza. Pošto je dužina niza 2^k, dužina periode mora biti stepen broja 2.

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. Dakle, za svaku poziciju i mora da važi da je $a_i = a_{i \bmod p}$. Na osnovu toga možemo jednostavno napraviti funkciju koja proverava da li je p perioda niza (za p koje deli n).

```
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;
}
```

Ocenimo složenost ovog algoritma. Složenost najgoreg slučaja funkcije `jePerioda` je $O(n)$ (međutim, često se može desiti da će se na razliku naići ranije, tako da je za očekivati da je složenost prosečnog slučaja bolja). Složenost najgoreg slučaja funkcije `minPerioda` nastupa kada niz nije tačno-periodičan, tj. kada se petlja u toj funkciji izvrši do kraja. Pošto se tokom izvršavanja petlje promenljiva `p` stalno uvećava duplo, kraj petlje se dostiže u $\log n$ koraka. Dakle, složenost najgoreg slučaja ovog rešenja je $O(n \log n)$.

U prethodnom algoritmu se vrši linearna pretraga niza mogućih eksponenata $[0, k]$. Jedno moguće poboljšanje je da se umesto linearne koristi binarna pretraga tog niza.

Razmotrimo sada i drugi, malo efikasniji način da se problem reši. 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 (jer niz nije $n/2$ periodičan, pa ne može biti periodičan ni za $n/4$, $n/8$, itd.). 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 (pošto je druga polovina niza jednaka prvoj, nju ne treba posebno analizirati). Ovim smo dobili induktivno rekurzivnu konstrukciju koja se završava kada je tekuća dužina niza 1 (njegova najmanja perioda je 1) ili kada se naiđe na niz čija prva polovina nije jednaka drugoj.

Možemo napraviti rekurzivnu implementaciju.

```
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);
}
```

Pošto je rekurzija repna, veoma jednostavno je se možemo osloboditi.

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

Dokaz korektnosti je prilično jednostavan, ali je analiza složenosti interesantnija. Rekurzivna implementacija zadovoljava rekurentnu jednačinu $T(n) = T(n/2) + O(n)$, čije je rešenje $O(n)$. To je jasno na osnovu master teoreme, ali lako se može videti i direktno. Naime, na prvom nivou se uradi $n/2$ poređenja unutar unutrašnje petlje, u drugom $n/4$, u trećem $n/8$, a u poslednjem pre izlaska iz rekurzije jedno poređenje (kada je niz dvočlan). Ukupno se uradi dakle $n/2 + n/4 + ... + 1$ koraka, što je $2^{k-1} + 2^{k-2} + ... + 1$, odnosno, na osnovu formule za zbir geometrijskog niza jednako $(2^k - 1)/(2 - 1)$ tj. $n - 1$.

I u ovom primeru smo demonstrirali da iako se u algoritmu javljaju ugnježene petlje, to ne povlači automatski složenost $O(n^2)$. Analizu je stoga potrebno sprovoditi veoma pažljivo.

Čak i kod najgoreg slučaja, razlika između složenosti $O(n)$ i $O(n \log n)$ nije prevelika, ali ipak može biti osetna. Niz koji se sastoji od $2^{21} - 1$ nule i jedne jedinice, koji očigledno nije periodičan, prvi algoritam obradi za oko $0,202$ sekunde, a drugi za oko $0,027$ sekundi, što je oko 10 puta brže - razlika je dosta manja nego u prethodnim primerima, ali ipak je nezanemariva. Sa druge strane, za niz iste dužine u kojem se periodično ponavljaju brojevi od 1 do 1024 prvi algoritam rezultat vraća za oko $0,050$ sekundi, a drugi za oko $0,030$ sekunde, što

je još manja razlika (generalno, što je perioda kraća, prvi algoritam će je brže detektovati, pri čemu čak iako je perioda dugačka, ali u okviru jedne periode nema puno samosličnosti, prvi algoritam će raditi prilično brzo).

Inkrementalnost

Još jedna tehnika u kojoj se izbegava vršenje istih izračunavanja iznova je *inkrementalnost*. Ilustrujmo je kroz nekoliko interesantnih primera.

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.

```
#include <iostream>

using namespace std;

int faktorijel(int k) {
  int p = 1;
  for (int i = 2; i <= k; i++)
    p *= i;
  return p;
}

int main() {
  int n;
  cin >> n;
  for(int k = 1; k <= n; k++)
    cout << faktorijel(k) << endl;
  return 0;
}
```

Za izračunavanje vrednosti $k!$ potrebno je $k-1$ množenja, pa je za izračunavanje vrednosti svih faktorijela do n potrebno $1+2+...+(n-1) = \frac{(n-1)n}{2}$ množenja, što je $O(n^2)$.

Mnogo bolje rešenje je da se primeti da je $k! = k \cdot (k-1)!$. 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 $k! = k \cdot (k-1)!$, za $k > 0$.

```
#include <iostream>

using namespace std;
```

```
int main() {
  int n;
  cin >> n;
  int p = 1;
  for(int k = 1; k <= n; k++) {
    p *= k;
    cout << p << endl;
  }
  return 0;
}
```

Pošto se svaki naredni od prethodnog dobija samo jednim množenjem, ukupan broj množenja je $n-1$ pa je složenost izračunavanja svih faktorijela $O(n)$.

Za algoritme kod kojih se neka vrednost ne izračunava stalno iz početka već se efikasno rekonstruiše na osnovu vrednosti poznatih u prethodnom koraku kažemo da su *inkrementalni*. Primetimo da se inkrementalnost veoma dobro uklapa u opšti mehanizam induktivno-rekurzivne konstrukcije, jer smo i u ovom slučaju na osnovu poznavanja rešenja za problem manje dimenzije uspeli direktno da odredimo rešenje za problem za jedan veće dimenzije. Inkrementalnost je tesno vezana za tehniku ojačavanja induktivne hipoteze (koju ćemo detaljnije ilustrovati kasnije). Naime, invarijanta petlje u funkciji `main` u prvom programu je da su ispisani svi faktorijeli zaključno sa $k-1$, dok je invarijanta petlje u funkciji `main` u drugom programu jača - ispisani su svi faktorijeli zaključno sa k, dok je $p = (k-1)!$.

Inkrementalnost je strašno važna tehnika za snižavanje vremenske složenosti algoritama i u nastavku ćemo je intenzivno koristiti.

Maksimalni zbir segmenta

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

Naglasimo da podniz može biti i prazan niz. Tako da je, u slučaju kada se niz sastoji samo od negativnih elemenata, segment sa najvećim zbirom upravo prazan segment.

Najdirektniji mogući način da se zadatak reši je da se izračuna zbir svih segmenata.

```
// tekuca vrednost maksimalnog zbira
int max_zbir = 0;
// proveravamo sve segmente odredjene pozicijama [i, j]
for (int i = 0; i < n; i++) {
  for (int j = i; j < n; j++) {
      // izracunavamo sumu segmenta [i, j]
      int z = 0;
```

```
      for (int k = i; k <= j; k++)
         z += a[k];
      // ako je dobijena suma veca od tekuce, azuriramo je
      if (z > max_zbir)
         max_zbir = z;
  }
}

cout << max_zbir << endl;
```

Korektnost prethodnog algoritma je veoma jednostavno dokazati. Maksimalni zbir je inicijalizovan na nulu (što je zbir praznog segmenta). Unutrašnja petlja računa zbir segmenta određenog indeksima $[i, j]$, dok dve spoljašnje petlje nabrajaju redom sve takve neprazne segmente, pa će svi segmenti biti obrađeni. Kada god se naiđe na segment čiji je zbir veći od tekućeg maksimalnog zbira, on se ažurira, tako da će na kraju promenljiva `max_zbir` sadržati maksimalnu vrednost zbirova svih segmenata. Ključni argument za korektnost algoritama koje do rešenja dolaze nabrajanjem i proverom svih mogućih kandidata za rešenje je to da su iscrpne tj. da se prilikom provere ni jedan od mogućih kandidata ne preskače. Kasnije ćemo videti da se efikasnost ovog tipa procedura često može poboljšati time što se za neke kandidate eksplicitno matematički pokaže da ne mogu da sadrže rešenje, pa se onda prilikom efektivne pretrage mogu preskočiti.

Pokušajte za vežbu da formulišete i precizne invarijante petlji i da formalno dokažete korektnost prethodnog algoritma. Ključni argument za korektnost je to da se tokom pretrage eksplicitno proveravaju baš svi segmenti niza.

Analizirajmo složenost ovog algoritma u terminima broja operacija sabiranja elemenata segmenata. Unutrašnja petlja se izvršava $j - i + 1$ puta (toliko puta se vrši operacija sabiranja), 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 $1 \cdot n + 2 \cdot (n - 1) + 3 \cdot (n - 2) + ... + (n - 1) \cdot 2 + n \cdot 1$. Dakle, segmenata dužine k ima $n - k + 1$, pa važi da je prethodni zbir jednak

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

Ovo je reda veličine $\frac{n^3}{2} - \frac{2n^3}{6} = \frac{n^3}{6}$, pa je ovaj algoritam veoma naivan i složenosti je $O(n^3)$.

Prilično je očigledno da se složenost 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.

```
// tekuca vrednost maksimalnog zbira
int max_zbir = 0;
// proveravamo sve segmente koji pocinju na poziciji [i]
for (int i = 0; i < n; i++) {
  // sumu racunamo inkrementalno
  int z = 0;
  // kraj segmenta ce biti na poziciji [j]
  for (int j = i; j < n; j++) {
    // u sumu uracunavamo element a[j]
    z += a[j];
    // ako je dobijena suma veca od maksimalne, azuriramo je
    if (z > max_zbir)
       max_zbir = z;
  }
}

cout << max_zbir << endl;
```

I u ovom slučaju proveravaju se eksplicitno baš svi segmenti, što je osnovni argument korektnosti procedure. Jedina razlika je što se za svaki novi segment zbir efikasnije računa, tako da je u dokazu korektnosti potrebno obrazložiti zašto će uvećanjem promenljive `z` za `a[j]` ona sadržati zbir svih elemenata niza u segmentu pozicija $[i, j]$ (što nije teško dokazati indukcijom, tj. kao invarijantu petlje).

Procenimo složenost ove implementacije. Za svako i unutrašnja petlja se izvršava $n - i$ puta i u svakom koraku se vrši jedno sabiranje. Dakle, u prvom koraku spoljne petlje imamo n sabiranja, u drugom $n - 1$ sabiranja, da bi se u poslednjem koraku vršilo samo jedno sabiranje. Ukupan broj sabiranja je $n + (n-1) + ... + 2 + 1 = \frac{n(n+1)}{2}$, pa smo korišćenjem inkrementalnosti sveli algoritam sa složenosti $O(n^3)$ na složenost $O(n^2)$.

Prvi algoritam, složenosti $O(n^3)$ nasumično generisan niz sa 10 hiljada elemenata između -3 i 3 obrađuje za $27,051$ sekundi, inkrementalna optimizacija to skraćuje na $0,074$ sekunde.

Određivanje broja rastućih segmenata datog niza brojeva

Problem: Dat je niz celih brojeva a. Definisati efikasan algoritam kojim se određuje koliko u tom nizu postoji rastućih segmenata dužine bar 2 i proceniti mu složenost. Segment $[i, j]$ $(0 \leq i < j < n)$ je rastući ako za njegove elemente važi $a_i < a_{i+1} < ... < a_j$.

Ovaj zadatak ponovo zahteva analizu segmenata niza, pa su moguća rešenja slična prethodnom zadatku u kom smo tražili segment maksimalnog zbira. Navodimo ga da bismo još jednom istakli značaj inkrementalnosti.

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 $a_k \leq a_{k-1}$ - ako takva dva elementa ne postoje, niz je rastući.

```
// ukupan broj rastucih segmenata
int brojRastucih = 0;
// proveravamo sve segmente odredjene pozicijama [i, j]
// posto je minimalna duzina segmenta 2,
// vrednost i ne sme ici dalje od n-2, a j krece od i+1
for (int i = 0; i < n - 1; i++) {
  for (int j = i + 1; j < n; j++) {
    // provera da li je segment a[i], a[i+1], ..., a[j] rastuci
    bool rastuci = true;
    for(int k = i + 1; k <= j && rastuci; k++)
      if (a[k] <= a[k-1])
        rastuci = false;
    // ako jeste, uvecavamo broj rastucih serija
    if (rastuci)
       brojRastucih++;
  }
}
```

Ovo rešenje je veoma neefikasno. Postoji $O(n^2)$ segmenata, a monotonost svakog se proverava algoritmom linearne složenosti, pa je ukupna složenost najgoreg slučaja $O(n^3)$. Precizna analiza složenosti je zapravo veoma slična onoj u prethodnom zadatku (ponovo se dobija veoma slična suma).

Naravno, do rešenja možemo efikasnije doći ako primenimo svojstvo inkrementalnosti tj. činjenicu da ako znamo da je segment $a_i, \ldots a_j$ rastući, tada je segment $a_i, \ldots, a_j, a_{j+1}$ 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.

```
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++;
```

Složenost ovog algoritma je $O(n^2)$.

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

$$\sum_{i=0}^{n-1} i \cdot a_i = 0 \cdot a_0 + 1 \cdot a_1 + 2 \cdot a_2 + 3 \cdot a_3 + \ldots + (n-1) \cdot a_{n-1}$$

u transformisanom nizu ima najveću vrednost.

I ovaj primer navodimo kako bismo ilustrovali tehniku inkrementalnosti i njen značaj.

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 jednostavno sami isprogramirati (pomoću jedne pomoćne promenljive i pomeranja svakog elementa za jedno mesto ulevo). Pošto broj operacija potreban za nezavisno izračunavanje svakog težinskog zbira linearno zavisi od dužine niza n, ovaj pristup dovodi do kvadratne složenosti algoritma tj. do složenosti $O(n^2)$.

```
long long tezinskiZbir(const vector<int>& a) {
   long long zbir = 0;
   for (int i = 0; i < a.size(); i++)
      zbir += i*a[i];
   return zbir;
}

int brojRotacijaZaMaksimalniZbir(const vector<int>& a) {
  // maksimum inicijalizujemo na težinski zbir pocetnog niza
  int maksBrojRotacija = 0;
  long long maksTezinskiZbir = tezinskiZbir(a);
  // n-1 puta ponavljamo
  for (int i = 1; i < n; i++) {
     // rotiramo niz za jedno mesto ulevo
     rotate(begin(a), next(begin(a)), next(begin(a), n));
     // izračunavamo novi težinski zbir
     long long zbir = tezinskiZbir(a);
     // ažuriramo maksimum ako je to potrebno
     if (zbir > maksTezinskiZbir) {
        maksBrojRotacija = i;
        maksTezinskiZbir = zbir;
     }
  }
  return maksBrojRotacija;
}
```

Na nasumično generisanom nizu od oko 20 hiljada elemenata, ova funkcija maksimum nalazi za oko $1,349$ sekundi.

Umesto efektivne rotacije svih elemenata niza (što može biti neefikasno zbog puno pomeranja 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 (kada brojač postane n vrednost mu se vraća na nulu što možemo postići bilo eksplicitnim ispitivanjem vrednosti nakon svakog uvećanja brojača, bilo izračunavanjem ostatka pri deljenju sa n, što može dosta usporiti program). Dobitak na brzini može biti

značajan, ali složenost i dalje ostaje $O(n^2)$. Funkcija implementirana na ovaj način, sa deljenjem po modulu n radi oko $1,062$ sekunde a sa ispitivanjem vraćanjem brojača na nulu kada dostigne n oko $0,490$ sekundi.

Još jedna tehnika kojom možemo izbeći rotaciju je da alociramo dvostruko više memorije, da elemente originalnog niza smestimo dva puta, jednom 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 $O(n^2)$. Naredni program na tekućem test-primeru radi oko $0,427$ sekundi.

```
// izracunava tezinski zbir n elemenata niza a od pozicije i
long long tezinskiZbir(const vector<int>& a, int i, int n) {
   long long zbir = 0;
   for (int j = 0; j < n; j++)
       zbir += a[i + j] * j;
   return zbir;
}

int brojRotacijaZaMaksimalniZbir(const vector<int>& a) {
   // ponavljamo elemente niza dva puta
   int n = a.size();
   a.resize(2*n);
   copy(begin(a), next(begin(a), n), next(begin(a), n));

   // maksimum inicijalizujemo na težinski zbir pocetnog niza
   int maksBrojRotacija = 0;
   long long maksTezinskiZbir = tezinskiZbir(a, 0, n);
   // računamo težinski zbir koji kreće od svake pozicije od i do n-1
   for (int i = 1; i < n; i++) {
      // računamo težinski zbir
      long long zbir = tezinskiZbir(a, i, n);
      // ako je potrebno ažuriramo podatke o maksumumu
      if (zbir > maksTezinskiZbir) {
        maksTezinskiZbir = zbir;
        maksBrojRotacija = i;
      }
  }

  // brišemo višak elemenata niza
  a.resize(n);

  return maksBrojRotacija;
}
```

Problem je moguće rešiti i efikasnije, ako se oslonimo na inkrementalni pristup. 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 težinski zbir dobijen prilikom pomeranja polaznog niza ulevo i puta, a sa z zbir svih elemenata niza. Prema tome važe sledeće jednakosti:

$$\begin{aligned}
z_0 &= 0 \cdot a_0 + 1 \cdot a_1 + 2 \cdot a_2 + ... + (n-2) \cdot a_{n-2} + (n-1) \cdot a_{n-1} \\
z_1 &= 0 \cdot a_1 + 1 \cdot a_2 + 2 \cdot a_3 + ... + (n-2) \cdot a_{n-1} + (n-1) \cdot a_0 \\
z_2 &= 0 \cdot a_2 + 1 \cdot a_3 + 2 \cdot a_4 + ... + (n-2) \cdot a_0 + (n-1) \cdot a_1 \\
&... \\
z_{n-1} &= 0 \cdot a_{n-1} + 1 \cdot a_0 + 2 \cdot a_1 + ... + (n-2) \cdot a_{n-3} + (n-1) \cdot a_{n-2}
\end{aligned}$$

$$z = a_0 + a_1 + ... + a_{n-2} + a_{n-1}$$

Primetimo da važi

$$\begin{aligned}
z_0 - z_1 &= a_1 + a_2 + ... + a_{n-1} - (n-1) \cdot a_0 = z - n \cdot a_0 \\
z_1 - z_2 &= a_2 + a_3 + ... + a_0 - (n-1) \cdot a_1 = z - n \cdot a_1 \\
&... \\
z_{n-2} - z_{n-1} &= a_{n-1} + a_0 + ... + a_{n-3} - (n-1) \cdot a_{n-2} = z - n \cdot a_{n-2}
\end{aligned}$$

itd.

Prema tome važi da je $z_{i-1} - z_i = z - n \cdot a_{i-1}$, tj.

$$z_i = z_{i-1} - z + n \cdot a_{i-1}.$$

Dakle, traženi težinski zbir posle pomeranja za jedno mesto ulevo možemo jednostavno izračunati bez pomeranja niza na osnovu poznatog zbira niza pre pomeranja i zbira svih elemenata niza.

Implementaciju možemo izvršiti na sledeći način. Izračunamo za polazni niz traženi težinski zbir $z_0 = \sum_{i=0}^{n-1} i \cdot a_i$ i zbir svih elemenata niza $z = \sum_{i=0}^{n-1} a_i$. Od svih zbirova treba izračunati najveći i odrediti posle koliko rotacija se taj najveći zbir postiže, što, naravno, radimo uobičajenim algoritmom određivanja pozicije maksimuma serije elemenata. Prvi izračunati zbir proglasimo najvećim, a broj pomeranja postavimo na 0. Zatim računamo tražene zbirove za nizove dobijene pomeranjem niza za jedno mesto ulevo i puta, i to redom za i od 1 do $n-1$. Traženi zbir z_i računamo tako što prethodni zbir z_{i-1} umanjimo za zbir svih elemenata z i uvećamo za $n \cdot a_{i-1}$. Proveravamo da li je dobijeni zbir veći od do sada najvećeg nađenog zbira i ako jeste korigujemo najveći zbir i broj pomeranja.

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. $O(n)$. Na tekućem test-primeru naredni program radi oko $0,012$ sekundi, što je neuporedivo efikasnije od svih prethodnih rešenja.

```
int brojRotacijaZaMaksimalniZbir(const vector<int>& a) {
  // izračunavamo zbir i težinski zbir početnog niza
  int tezinskiZbir = 0;
  int zbir = 0;
  for (int i = 0; i < n; i++) {
    tezinskiZbir += i*a[i];
    zbir += a[i];
  }
  // najveci do sada vidjeni tezinski zbir
  int maksTezinskiZbir = tezinskiZbir;
  // broj rotacija kojima se on postiže
  int maksBrojRotacija = 0;
  for (int i = 1; i < n; i++) {
    // ažuriramo težinski zbir
    tezinskiZbir = tezinskiZbir - zbir + n * a[i-1];
    // ažuriramo maksimum ako je to potrebno
    if (tezinskiZbir > maksTezinskiZbir) {
      maksTezinskiZbir = tezinskiZbir;
      maksBrojRotacija = i;
    }
  }
  return maksBrojRotacija;
}
```

Postavljanje rutera

Problem: Duž jedne ulice su ravnomerno 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 (računati samo dužinu kablova od zgrade do zgrade i zanemariti dužine unutar zgrada). Na primer, ako je broj korisnika po zgradama jednak $3, 5, 1, 6, 2, 4$, ruter treba postaviti u četvrtu zgradu sleva i dužina kablova je tada jednaka $3 \cdot 3 + 2 \cdot 5 + 1 \cdot 1 + 1 \cdot 2 + 2 \cdot 4 = 30$.

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

$$\sum_{i=0}^{n-1} |k - i| \cdot a_i,$$

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

$$d_k = \sum_{i=0}^{k-1}(k-i)\cdot a_i + \sum_{i=k+1}^{n-1}(i-k)\cdot a_i.$$

Ako je ruter na zgradi $k+1$, tada je dužina kablova jednaka

$$d_{k+1} = \sum_{i=0}^{k}(k+1-i)\cdot a_i + \sum_{i=k+2}^{n-1}(i-k-1)\cdot a_i.$$

Razlika između te dve sume jednaka je

$$\begin{aligned}
d_{k+1} - d_k &= \left(\sum_{i=0}^{k}(k+1-i)\cdot a_i - \sum_{i=0}^{k-1}(k-i)\cdot a_i\right) + \left(\sum_{i=k+2}^{n-1}(i-k-1)\cdot a_i - \sum_{i=k+1}^{n-1}(i-k)\cdot a_i\right) \\
&= \left(\sum_{i=0}^{k-1}((k+1-i)-(k-i))\cdot a_i\right) + a_k - a_{k+1} + \left(\sum_{i=k+2}^{n-1}((i-k-1)-(i-k))\cdot a_i\right) \\
&= \sum_{i=0}^{k-1} a_i + a_k - a_{k-1} - \sum_{i=k+2}^{n-1} a_i \\
&= \sum_{i=0}^{k} a_i - \sum_{i=k+1}^{n-1} a_i
\end{aligned}$$

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 $k+1$. To je zapravo intuitivno prilično jasno i bez prethodnog komplikovanog matematičkog izvođenja. 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. Ovaj primer lepo pokazuje kako se intuicija i formalno matematičko izvođenje prepliću - kada možemo do rešenja doći intuitivnim putem, ono je obično veoma elegantno i jednostavno. Sa druge strane, ako ne vidimo odmah rešenje problema, matematički aparat nam može pomoći da do njega dođemo.

Ukupne brojeve stanara pre i posle date zgrade možemo takođe računati inkrementalno (pri prelasku na narednu zgradu, prvi broj se uvećava, a drugi umanjuje za broj stanara tekuće zgrade).

Dakle, u programu možemo da pamtimo tri stvari: dužinu kablova d_k ako je ruter na poziciji k, ukupan broj stanara pre_k pre zgrade k i ukupan broj stanara $posle_k$ od zgrade k do kraja. Na početku, kada je $k = 0$, prvi broj d_0 moramo eksplicitno izračunati kao $\sum_{i=1}^{n-1} i \cdot a_i$ (za to nam je potrebno vreme $O(n)$), drugi broj treba inicijalizovati na nulu $pre_0 = 0$, a treći na ukupan broj svih stanara $posle_k = \sum_{i=0}^{n-1} a_i$ (i za to nam je potrebno vreme $O(n)$). Zatim za svako k od 1 do $n - 1$ računamo $pre_k = pre_{k-1} + a_{k-1}$, $posle_k = posle_{k-1} - a_{k-1}$ i $d_k = d_{k-1} + pre_k + posle_k$.

```
// krećemo od zgrade 0
// ukupna dužina kablova ako je ruter u tekućoj zgradi
long long duzina_kablova = 0;
for (int i = 0; i < n; i++)
  duzina_kablova += stanara[i] * i;
// broj stanara pre tekuće zgrade
long long stanara_pre = 0;
// broj stanara od tekuće zgrade do kraja
long long stanara_posle = 0;
for (int i = 0; i < n; i++)
  stanar_posle += stanara[i];

// minimalna duzina kablova
long long min_duzina_kablova = duzina_kablova;

// obrađujemo sve zgrade od 1 do n-1
for (int k = 1; k < n; k++) {
   // ažuriramo brojeve stanara
   stanara_pre += stanara[k-1];
   stanara_posle -= stanara[k-1];
   // ažuriramo duz
   duzina_kablova += stanara_pre - stanara_posle;
   if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
}

cout << min_duzina_kablova << endl;
```

Ukupno vreme izvršavanja ovog algoritma je linearno (izvršavaju se tri petlje, svaka složenosti $O(n)$). Recimo i da nismo morali održavati broj stanara od zgrade k do kraja, već smo ga mogli svaki put izračunavati kao razliku između ukupnog broja stanara i broja stanara pre zgrade k (to praktično ne bi uticalo na vreme izvršavanja).

Izbegavanje nepotrebnih izračunavanja

Algoritam bi trebalo da izračuna samo one stvari koje su mu neophodne da bi vratio traženi rezultat. Višak posla obično dovodi do loše efikasnosti algoritma. Ilustrujmo ovo kroz nekoliko primera.

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.

```
int drugiNaListi(int a[], int n) {
  sort(a, a+n, greater<int>());
  return a[1];
}
```

Složenost ovog rešenja dolazi od složenosti sortiranja i iznosi $O(n \log n)$.

Razmislite koliko se nepotrebno vremena troši nizu od 1000 elemenata da se ustanovi poredak 998 elemenata iza prva dva. 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];
}
```

Iako se u implementaciji javljaju dve ugnežđene petlje, složenost ovog algoritma je $O(n)$, jer se spoljna petlja izvršava samo dva puta (ukupan broj poređenja je $n + (n - 1)$).

Ipak, najbolje rešenje koristi specijalizovani algoritam.

```
int drugiNaListi(int a[], int n) {
  // prvi i drugi maksimum inicijalizujemo na -1
  int prviMax, drugiMax;
  prviMax = drugiMax = -1;

  for(int i = 0; i < n; i++) {
    // ako je potrebno ažuriramo vrednost maksimuma
    if (a[i] > prviMax)  {
      drugiMax = prviMax;
      prviMax  = a[i];
    } else if (a[i] > drugiMax) {
      drugiMax = a[i];
    }
  }
```

```
  // vraćamo vrednost drugog maksimuma
  return drugiMax;
}
```

Uopštenjem ovog problema (nalaženjem k-tog po veličini elementa bavićemo se kasnije).

Morzeov niz

Problem: Niz

$$1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, \ldots,$$

koji se sastoji od nula i jedinica, gradi se na sledeći način: prvi element je 1; drugi se dobija logičkom negacijom prvog $!1 = 0$, treći i četvrti logičkom negacijom prethodna dva $!1 = 0$, $!0 = 1$, peti, šesti, sedmi i osmi logičkom negacijom prva četiri - dobija se $0, 1, 1, 0$ itd. Dakle, krenuvši od jednočlanog segmenta 1, svakom početnom segmentu koji je dužine 2^k (k uzima vrednosti $0, 1, 2, ...$) 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 ugnježdena petlja gde se u svakom unutrašnjem koraku duplira dužina niza.

```
int morzeov(int n) {
   // rezervisemo prostor za (n+1) elemenata, da bi mogli da izracunamo
   // element na poziciji n
   vector<bool> m(n+1);
   // prvi element niza
   m[1] = true;
   // st je trenutna duzina niza i bice jednaka stepenu dvojke
   // u pocetku st = 2^0
   int st = 1;
   while (st < n) {
      // popunjavamo "drugu" polovinu niza
      for (int i = st+1; i <= st+st && i <= n; i++)
        m[i] = !m[i-st];
      // dupliramo duzinu niza (naredni stepen dvojke)
      st += st;
   }
   return m[n];
}
```

Umesto dvostruke, možemo upotrebiti i jednostruku petlju.

```
int morzeov(int n) {
   vector<bool> m(n+1);
   m[1] = true;
   int k = 1;
   for (int i = 2; i <= n; i++) {
      m[i] = !m[i - k];
      if (i == k * 2)
         k *= 2;
   }
   return m[n];
}
```

Složenost ovog pristupa je očigledno $O(n)$ (kako vremenska, tako i memorijska). Prethodno rešenje podrazumeva formiranje svih elemenata niza koji prethode n-tom članu. Da bi se izračunao 2001. član, mora se izračunati svih 2000 prethodnih članova, međutim, jasno je da je to suvišno jer svaki element negiranog prethodnog segmenta direktno zavisi samo od jednog elementa tog prethodnog segmenta, a ne od svih njih, tako da za njegovo određivanje nije potrebno poznavanje celokupnog prethodnog segmenta, već samo jednog njegovog karakterističnog elementa.

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 $m_n = !\, m_{n-2}$. Nakon toga dobijamo da je $m_5 = !\, m_1$, $m_6 = !\, m_2$, $m_7 = !\, m_3$ i $m_8 = !\, m_4$. U ovom slučaju važi da je $m_n = !\, m_{n-4}$.

Dakle, 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,

$$m_{15} = !\, m_{15-8} = !\, m_7 = !\, (!\, m_{7-4}) = m_{7-4} = m_3 = !\, m_{3-2} = !\, m_1 = !\, 1 = 0.$$

Traženi broj se dobija u malom broju koraka i za veće vrednosti broja n. Na primer, za $n = 2001$ važi sledeće.

$$m_{2001} = !\, m_{2001-1024} = m_{977-512} = !\, m_{465-256} = m_{209-128} = !\, m_{81-64} = m_{17-16} = 1.$$

Rekurentna formula nam ukazuje na to da rešenje možemo veoma jednostavno realizovati uz pomoć rekurzivne funkcije.

```
// vraca najveci stepen od 2, koji je strogo manji od n
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 $O(\log n)$, jer se promenljiva `max` u svakom koraku povećava duplo, sve dok ne dostigne ili prestigne vrednost $n/2$.

U svakom novom pozivu funkcije `morzeov` skida se jedan bit broja n, pa je broj rekurzivnih poziva ograničen brojem bitova broja n, što je $O(\log n)$. Složenost je u najgorem slučaju $O(\log^2(n))$.

Do rešenja možemo doći i iterativno.

```
int morzeov(int n) {
  int m = 1;
  while (n > 1) {
    n -= maxStepen2(n);
    m = !m;
  }
  return m;
}
```

Odsecanje u pretrazi

Jako često su algoritmi zasnovani na pretrazi tj. na proveri različitih kandidata za rešenje (ili kandidata za kontraprimer). Računari veoma brzo nabrajaju stvari, ali nekada je kandidata previše i provera svih njih može dovesti do neefikasnog algoritma. Na sreću, često se matematički može pokazati da se rešenje ne može nalaziti u nekom skupu kandidata i pretraga se može ubrzati time što se onda ti kandidati prosto izostave. Naravno, takvi algoritmi uvek moraju biti praćeni (makar neformalnim) dokazom korektnosti koji nam daju opravdanje na osnovu kog smo pretragu sasekli. Ilustrujmo ovo kroz nekoliko interesantnih primera.

Ispitivanje da li je broj prost

Problem: Definisati efikasnu funkciju koja proverava da li je broj prost. Proceni joj složenost.

Naivno rešenje je zasnovano na linearnoj proveri svih delilaca.

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

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.

```
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;
}
```

Složenost najgoreg slučaja ovog algoritma je $O(\sqrt{n})$.

Prvi algoritam utvrđuje da je broj 1000000007 prost nakon oko $3,171$ sekundi, a drugi nakon oko $0,005$ sekundi. Oba algoritma, međutim, utvrđuju da broj 1000000008 nije prost nakon oko $0,004$ sekunde.

Ključni razlog uštede je to što smo odsekli neke slučajeve u pretrazi pozivajući se na teoremu koja garantuje da je provera tih slučajeva nepotrebna, jer su pokriveni drugim slučajevima. Ovakvo odsecanje se veoma često koristi kao način popravljanja efikasnosti algoritama.

Primetimo da postoje mnogo efikasniji algoritmi za rešavanje ovog problema, kao i da se prethodna implementacija može malo ubrzati. Na osnovu činjenice da su svi prosti brojevi (pa samim tim i prosti delioci) oblika $6k+1$ ili $6k+5$ (što je jednako $6t-1$, za $t = k+1$), [1] petlja se može organizovati tako što se brojač k množi sa 6, a u jednom prolasku petlje se ispituju dva kandidata. Primetimo da 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.

[1] Proizvoljan broj se može zapisati u obliku $n = 6k + j$, za $0 \leq j < 6$. Brojevi oblika $6k$, $6k+2$, $6k+3$ i $6k+4$ su deljivi brojem 2 ili 3, pa prosti brojevi mogu biti zapisani samo u obliku $6k+1$ ili $6k+5$.

```
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;
}
```

Ovo je primer optimizacije koja ne menja asimptotsku složenost, već ubrzava algoritam samo za konstantni faktor. Naime, broj koraka je takav da je otprilike $36k^2 > n$ tj. oko $\frac{\sqrt{n}}{6}$, međutim, u svakom koraku se proveravaju dve deljivosti, pa je broj provera oko $\frac{\sqrt{n}}{3}$, te možemo očekivati ubrzanje najgoreg slučaja oko tri puta. Asimptotska složenost je i dalje $O(\sqrt{n})$.

Eratostenovo sito

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. Proceni joj složenost.

Direktan način je da za svaki broj pozovemo funkciju iz prethodnog zadatka.

```
void sviProsti(bool prost[], int n) {
  for (int i = 0; i <= n; i++)
    prost[i] = jeProst(i);
}
```

Pošto je složenost najgoreg slučaja prethodne funkcije $O(\sqrt{n})$ i ona se poziva $O(n)$ puta, složenost ove implementacije je $O(n\sqrt{n}) = O(n^{3/2})$.

Bolji način da se zadatak reši je da se upotrebi Eratostenovo sito u kom se redom precrtavaju svi umnošci prostih brojeva. Postupak zahteva niz dužine n (u implementaciji dužine $n + 1$ jer je niz indeksiran od 0) 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 (brojevi se "prosejavaju kroz sito").

```
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;
}
```

Primetimo da je i ovde pretraga sasečena tj. da se i u ovom slučaju iteracija vršila samo do $\sqrt{n}$, što značajano ubrzava ceo postupak, a opravdano je na osnovu istog argumenta kao i u slučaju ispitivanja da li je pojedinačan broj prost.

Dokažimo korektnost ovog algoritma.

Lema: Invarijanta spoljašnje petlje je da je $2 \leq d \leq \lfloor\sqrt{n}\rfloor + 1$ i da su precrtani 0, 1 i tačno svi oni brojevi manji ili jednaki n koji imaju prave delioce u intervalu $[2, d)$ (tj. za svaki takav broj x važi da je u nizu `prost` zapisano da taj broj nije prost, dok je za sve brojeve koji nemaju delilaca manjih od d zapisano da su prosti).

- 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 $d \leq \lfloor\sqrt{n}\rfloor$. Onda važi i $d + 1 \leq \lfloor\sqrt{n}\rfloor + 1$, odnosno $d' \leq \lfloor\sqrt{n}\rfloor + 1$.

  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, na osnovu pretpostavke znamo da njega deli neki delilac t iz intervala $[2, d)$ (jer je sigurno različit od 0 i 1). Svi brojevi manji ili jednaki od n koji imaju delioce iz intervala $[2, d)$ su na osnovu pretpostavke su 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 brojevi $2d$, $3d$, …tj. svi brojevi koji su manji ili jednaki od n i deljivi su brojem d, osim broja d (njemu d nije pravi delilac). Ovim se eksplicitno invarijanta proširuje i na njih i ostaje održana.

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

Pošto se petlja završila, važi da je $d > \lfloor\sqrt{n}\rfloor$, a pošto na osnovu invarijante znamo da je $d \leq \lfloor\sqrt{n}\rfloor + 1$, važi da je $d = \lfloor\sqrt{n}\rfloor + 1$. 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 $x \leq n$ 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 (ponovo na osnovu teoreme na osnovu koje znamo da ako broj x ima pravog delioca iz intervala $[\sqrt{x}, x)$ onda ima i pravog delioca iz intervala $[2, \sqrt{x}]$, pa neprecrtani brojevi nemaju pravih delioca).

Analiza složenosti je komplikovanija i zahteva određeno (doduše veoma elementarno) poznavanje teorije brojeva. Procenimo broj izvršavanja tela unutrašnje petlje. U početnom koraku spoljne petlje precrtava se oko $\frac{n}{2}$ elemenata. U narednom, oko $\frac{n}{3}$. U narednom koraku je broj 4 već precrtan, pa se ne precrtava ništa. U narednom se precrtava oko $\frac{n}{5}$, nakon toga opet ništa, zatim $\frac{n}{7}$ itd. U poslednjem koraku se precrtava oko $\frac{n}{\sqrt{n}}$ elemenata. Dakle, broj precrtavanja je

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

Još je veliki Ojler otkrio da je zbir $H(m) = 1 + 1/2 + 1/3 + ... + 1/m = \sum_{d \leq m} \frac{1}{d}$ (takozvani harmonijski zbir) asimptotski jednak $\log m$ (razlika između ove dve funkcije teži takozvanoj Ojler-Maskeronijevoj konstanti $\gamma \approx 0.5772156649$) - samim tim znamo da taj zbir divergira. Takođe, otkrio je da kada se sabiranje vrši samo po prostim brojevima, tada se zbir ponaša kao logaritam harmonijskog zbira, tj. kao $\log \log m$ (pa je i on divergentan). Dakle, u našem primeru možemo zaključiti da je broj precrtavanja jednak $n \cdot \log \log \sqrt{n}$. Pošto je $\log \log \sqrt{n} = \log \log n^{\frac{1}{2}} = \log \left(\frac{1}{2} \log n \right) = \log \frac{1}{2} + \log \log n$, važi da je složenost Eratostenovog sita $O(n \cdot \log \log n)$. Iako nije linearna, funkcija $\log \log n$ toliko sporo raste, da se za sve praktične potrebe Eratostanovo sito može smatrati linearnim u odnosu na n (što je opet dosta sporije samo od ispitivanja da li je broj n prost, što ima složenost $O(\sqrt{n})$).

Brojanje prostih brojeva manjih od 10^7 (ima ih 664579) korišćenjem zasebne provere svakog od brojeva i osnovne implementacije provere da li je broj prost traje oko $4,718$ sekundi, a korišćenjem ubrzane implementacije koja proverava samo brojeve oblika $6k - 1$ i $6k + 1$ traje oko $1,609$ sekundi. Sa druge strane, Eratostenovo sito traje samo oko $0,143$ sekunde.

Binarna pretraga

Sprovođenje binarne umesto linearne pretrage takođe je optimizacija koja spada u kategoriju odsecanja prilikom pretrage. Naime, zahvaljujući svojstvu uređenosti elemenata niza, znamo da neke delove niza nema potrebe pretraživati, jer ne mogu da sadrže rešenje.

Problem: Proveriti da li u strogo rastuće sortiranom nizu brojeva postoji neka pozicija i takva da je $a_i = i$. Ako postoji vratiti najmanju takvu, a ako ne postoji, vratiti -1.

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

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

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 $a_{i+1} \geq a_i + 1$. Slično je $a_{i+2} > a_{i+1}$, pa je $a_{i+2} \geq a_{i+1} + 1 \geq a_i + 2$. Nastavljanjem ovog rezona važi da je $a_j \geq a_i + j$ (formalno, ovo je moguće dokazati indukcijom). 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) {
  // sprovodimo binarnu pretragu - ako trazeni element a[i] = i
  // postoji, on se nalazi u intervalu [l, d]
  // invarijanta:
  //    za sve elemente u intervalu [0, l) važi da je a[i] < i
  //    za sve elemente u intervalu (d, n) važi da je a[i] >= i
  int l = 0, d = n-1;
  // dok interval [l, d] nije prazan
  while (l <= d) {
    // pronalazimo sredinu intervala
    int s = l + (d - l) / 2;
    if (a[s] < s)
      // najmanji element takav da je a[i]=i može biti samo desno od s
      l = s + 1;
    else
      // najmanji element takav da je a[i]=i može biti samo levo od s
      d = s - 1;
  }
  // svi elementi levo od l su takvi da je a[l] < l
  // ako postoji element takav da je a[l] = l, najmanji takav može
  // biti jedino na poziciji i
  if (l < n && a[l] == l)
    return l;
  else
    return -1;
}
```

Prodiskutujmo binaru pretragu iz ugla odsecanja. Naime, zahvaljujući činjenici da je niz $a_i - i$ sortiran, kada za neko s utvrdimo da je $a_s < s$, znamo da se najmanje i takvo da je $a_i = i$ ne može nalaziti levo od pozicije s (niti na njoj). Zbog toga možemo da preskočimo (isečemo) pretragu svih pozicija levo od s i da postavimo promenljivu `l` na vrednost $s + 1$. Slično, ako je $a_s \geq s$ onda se

najmanja pozicija takva da je $a_i = i$ ne može nalaziti na pozicijama iza s, pa možemo da preskočimo (isečemo) pretragu svih pozicija desno od s i postavimo promenljivu `d` na vrednost $s + 1$.

Maksimalni zbir segmenta

Vratimo se problemu određivanja vrednosti maksimalnog zbira nekog segmenta niza. Prethodna dva algoritma koja smo prikazali se mogu slobodno nazvati trivijalnim, jer se do njih dolazi prilično direktno i veoma jednostavno im se i dokazuje korektnost i analizira složenost. Međutim, oni su prilično neefikasni za rešavanje ovog problema. Značajno unapređenje možemo dobiti kada primetimo da veliki broj segmenata uopšte ne moramo da obrađujemo, jer iz nekih drugih razloga znamo da njihov zbir ne može biti maksimalan.

Posmatrajmo niz `-2 3 2 -3 -3 -2 4 5 -8 3` i zbirove svih njegovih nepraznih segmenata (u i-tom redu i j-toj koloni se nalazi zbir segmenta $[i, j]$).

```
-2  1   3   0  -3  -5  -1   4  -4  -1
    3   5   2  -1  -3   1   6  -2   1
        2  -1  -4  -6  -2   3  -5  -2
           -3  -6  -8  -4   1  -7  -4
               -3  -5  -1   4  -4  -1
                   -2   2   7  -1   2
                        4   9   1   4
                            5  -3   0
                               -8  -5
                                    3
```

Razmotrimo bilo koji segment koji počinje negativnim brojem. Takav segment onda ne može imati maksimalni zbir, pošto se izostavljanjem tog prvog negativnog broja dobija veći zbir. Ovo zapažanje se može uopštiti. Ukoliko niz počinje prefiksom negativnog zbira, iz istog razloga, nijedan segment čiji je on prefiks ne može imati maksimalni zbir. Otud, pri inkrementalnom proširivanju intervala udesno, čim se ustanovi da je tekući zbir negativan, moguće je prekinuti dalje proširivanje.

Na primer, čim vidimo da je prvi element `-2`, možemo prekinuti dalju obradu elemenata prve vrste, jer će svi elementi druge vrste sigurno biti za dva veći nego odgovarajući elementi prve vrste. Slično, u drugoj vrsti, kada se prilikom proširivanja segmenta (koji počinje na drugoj poziciji) dobije negativan parcijalni zbir `-1`, možemo zaključiti da će segmenti u ostatku niza (iz koga je izbačen taj segment) imati veći parcijalni zbir. Onda možemo prekinuti sa obradom dužih segmenata koji počinju na toj poziciji, jer smo sigurni da će za svaki od njih kasnije postojati segment sa većim zbirom koji se dobija izostavljanjem prefiksa `3 2 -3 -3` (jer je njegov zbir `-1`). Zaista, od preostalih zbirova `-3 1 6 -2 1` u drugoj vrsti za jedan su veći zbirovi `-2 2 7 -1 2` u šestoj vrsti koji su dobijeni izostavljanjem tog prefiksa. Obratimo pažnju da prekid unutrašnje petlje na

ovaj način uzrokuje da se maksimalna vrednost u tekućoj vrsti ne mora uopšte naći. Petlja koja obrađuje drugu vrstu će biti prekinuta kada je tekuća vrednost maksimuma 5 iako je maksimum te vrste 6. Sigurni smo da će u nekoj narednoj vrsti postojati veća vrednost od te najveće (zaista, u šestoj vrsti se javlja 7), pa nam nalaženje stvarnog maksimuma u tekućoj vrsti uopšte nije neophodno.

```
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++;
}
```

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. Primetimo da u tom slučaju maksimalni zbir biva nađen za $i = 0$ i $j = n-1$. Nakon toga se, uvećavanjem indeksa i, zbir smanjuje pošto se svakim skraćivanjem segmenta sleva izostavlja neki pozitivan broj koji doprinosi zbiru. I ovo zapažanje se može uopštiti. Ne samo što je nepoželjno skratiti interval sleva za neki pozitivan broj, već je nepoželjno skratiti ga za bilo koji prefiks čiji je zbir pozitivan. Pitanje je dokle takvi prefiksi sežu? Bar do elementa čijim obuhvatanjem dobijamo prvi negativan prefiks. Otud segment maksimalnog zbira ne može počinjati ni na jednoj poziciji između tekuće početne pozicije i prve pozicije na kojoj zbir postaje negativan.

Na primer, u navedenom primeru maksimalni segment ne može počinjati na trećoj poziciji, jer se proširivanjem nalevo i dodavanjem elementa 3 sa druge pozicije dobijaju sigurno zbirovi koji su veći za tri. Elementi druge vrste su za 3, 5 tj. 2 su veći od odgovarajući elemenata treće, četvrte i pete vrste, pa te tri vrste uopšte nema potrebe razmatrati.

Zahvaljujući ovom zapažanju, nije neophodno uvećavati promenljivu i za jedan, već je moguće nastaviti iza elementa čijim je uključivanjem suma postala negativna.

```
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;
}
```

Ocenimo složenost ovog algoritma. Neko bi naivno mogao pomisliti da će složenost algoritma biti bar kvadratna, jer se u programu javljaju dve ugnežđene petlje. Međutim, prilično lako možemo pokazati da je složenost ovog algoritma $O(n)$. Naime, lako se primećuje da se vrednost obe brojačke promenljive i i j stalno uvećava i nikada ne smanjuje. Naime, kada se prekine iteracija unutrašnje petlje, promenljiva i se postavlja na $j + 1$ i nova runda unutrašnje petlje kreće od $j + 1$, što je veće od vrednosti j na kojoj se u prethodnoj rundi stalo. Pošto se postupak sigurno prekida kada obe promenljive dostignu vrednost n, one se mogu uvećati najviše $2n$ puta, tako da je složenost $O(n)$.

Algoritme u kojima održavamo dve pozicije koje se kroz niz kreću samo u jednom smeru nazivamo tehnikom dva pokazivača i oni su po pravilu linearne složenosti (ovaj algoritam je tog tipa).

Ključni način dobijanja na efikasnosti u ovom algoritmu je isti kao i u slučaju ispitivanja prostih brojeva - izvršili smo odsecanje pretrage, tj. na osnovu matematičkih argumenata eliminisali smo potrebu da tokom pretrage proveravamo mnoge slučajeve. Tu tehniku ćemo intenzivno koristiti i u nastavku.

Podsetimo se, prvi algoritam, složenosti $O(n^3)$ nasumično generisan niz sa 10 hiljada elemenata između -3 i 3 obrađuje za $27,051$ sekundi, inkrementalna optimizacija to skraćuje na $0,074$ sekunde, dok se odsecanjem vreme spušta na $0,004$ sekunde.

Pretprocesiranje radi efikasnije pretrage

U nekim slučajevima je potrebno izvršiti određenu vrstu pretprocesiranja, da bi se u kasnijoj fazi algoritam mogao efikasnije izvršiti. Razmotrimo nekoliko primera.

Provera duplikata u nizu

Problem: Definisati funkciju koja efikasno proverava da li u datom nizu celih brojeva ima duplikata i proceni joj složenost.

Naivni način je da se ispita svaki par elemenata.

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

Složenost najgoreg slučaja ovog pristupa je prilično očigledno kvadratna (parova ima $\binom{n}{2}$). U najgorem slučaju (kada nema duplikata) u prvom koraku spoljašnje petlje vrši se $n-1$ koraka unutrašnje petlje, u drugom $n-2$ koraka, itd. Ukupno se, dakle, vrši $(n-1)+(n-2)+...+1$ koraka, što je opet $\frac{n(n-1)}{2}$, te je složenost $O(n^2)$.

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.

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

Sortiranje smo sproveli korišćenjem bibliotečke funkcije i njena složenost je $O(n \log n)$. Nakon toga, pretraga se vrši u vremenu $O(n)$. Dakle, ukupna složenost je veća od ove dve, a to je $O(n \log n)$. Videćemo kasnije da zadatak možemo rešiti i pomoću specijalizovanih struktura podataka u istoj asimptotskoj složenosti, međutim, direktna rešenja, sa običnim nizovima su obično efikasnija (za konstantni faktor).

Za niz koji sadrži redom sve brojeve od 0 do 99999 prvi algoritam utvrđuje da duplikata nema za oko $2,042$ sekunde, a drugi za oko $0,012$ sekundi.

Ključni dobitak na efikasnosti u ovom primeru je usledio ponovo time što je na osnovu matematičkog argumenta eliminisana potreba za proverom velikog broja slučajeva. Međutim, za razliku od prethodnih primera gde je to učinjeno na osnovu analize originalnog ulaza, ovde je ulaz morao biti preprocesiran tako da odsecanje bude moguće. Najčešći oblik pretprocesiranja koji dopušta različite oblike kasnijih odsecanja prilikom analize elemenata niza je sortiranje. Veoma interesantan savet prilikom dizajna algoritama je "ako ne znaš odakle da kreneš, probaj da sortiraš". Zbog njihovog značaja primenom sortiranja i binarne pretragom ćemo se baviti posebno u jednoj od narednih glava.

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. Naime, zbir segmenta određenog pozicijama iz intervala $[l, d]$ jednak je:

$$\sum_{i=l}^{d} a_i = \sum_{i=0}^{d} a_i - \sum_{i=0}^{l-1} a_i$$

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;
// zbirPrefiksa[i] - zbir elemenata iz intervala [0, i)
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;
}
```

Pretprocesiranje je u ovom primeru podrazumevalo izračunavanje pomoćnih podataka koji su omogućili efikasnije kasnije izvršavanje algoritma. Prefiksni zbirovi su tehnika koja se često koristi prilikom izgradnje efikasnih algoritama i još primera njene upotrebe biće dato kasnije.

Čas 3.1, 3.2, 3.3 - Primeri induktivno-rekurzivne konstrukcije

Rešavanje problema rekurzijom

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. Na primer 00, 01, 11, 10 je Grejov kôd dužine 2^2. Zaista 00 i 01 se razlikuju samo na drugom bitu, 01 i 11 samo na prvom, 11 i 10 samo na drugom, a 10 i 00 samo na prvom bitu. Definisati funkciju koja konstruiše Grejov kôd dužine 2^n za proizvoljno n.

Posmatrajmo nekoliko Grejovih kodova. Grejov kôd dužine 2^0 sadrži samo praznu nisku, dužine 2^1 je 0, 1, dužine 2^2 je 00, 01, 11, 10, dužine 2^3 je 000, 001, 011, 010, 110, 111, 101, 100 itd. Razmotrimo kako je dobijen Grejov kôd dužine 2^3. Prva 4 elementa su dobijena tako što je ispred odgovarajućeg elementa u Grejovom kodu dužine 2^2 dodata nula. Naredna 4 elementa su dobijena tako što je ispred elemenata u Grejovom kodu dužine 2^2 postavljena cifra 1, a onda su tako dobijeni elementi poređani u obratnom redosledu. Ovo je 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 indukcijom. Bazu indukcije čini kôd dužine 2^0 - u njemu postoji jedan zapis dužine 0, nema susednih zapisa, pa je kôd trivijalno Grejov. Pretpostavimo da zapisi g_0, g_1, ..., $g_{2^{n-1}-1}$ čine Grejov kôd dužine 2^{n-1}. Posmatrajmo niz zapisa $0g_0$, $0g_1$, ..., $0g_{2^{n-1}-1}$, $1g_{2^{n-1}-1}$, ..., $1g_1$, $1g_0$ i dokažimo da on čini Grejov kôd. Svaka dva susedna zapisa oblika $0g_i$ i $0g_{i+1}$ se razlikuju samo na jednom bitu, jer im je prvi bit isti, a po induktivnoj pretpostavci se g_i i g_{i+1} razlikuju tačno na jednom bitu. Slično važi i za zapise $1g_i$ i $1g_{i-1}$. Susedni su još zapisi $0g_{2^{n-1}-1}$ i $1g_{2^{n-1}-1}$ i oni se razlikuju samo na prvom bitu i zapisi $1g_0$ i $0g_0$ (poslednji i prvi zapis) i oni se razlikuju samo na prvom bitu. Dakle svaka dva susedna zapisa se razlikuju samo na jednom bitu, pa je kôd Grejov.

Definišimo funkciju koja određuje k-ti po redu zapis Grejovog koda dužine 2^n (podrazumevaćemo da je $0 \leq k < 2^n$). 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 elemet Grejovog koda dužine 2^{n-1} dopunjen početnom jedinicom (jer se zapisi u ovom delu redaju obratnim redosledom, pa se, na primer, za $k = 2^{n-1}$ vraća zapis na poziciji $2^n - 1 - 2^{n-1} = 2^{n-1} - 1$, tj. poslednji zapis u manjem Grejovom kodu, a za $k = 2^n - 1$ vraća zapis na poziciji 0 tj. prvi zapis u manjem Grejovom kodu).

Izračunavanje stepena dvojke trivijalno se vrši bitovskim operacijama.

```
string grej(int n, int k) {
   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);
}
```

Primetimo da je ova funkcija prilično efikasna, jer nam za izračunavanje Grejovog koda dužine 2^n na poziciji k nije potrebno poznavanje celog Grejovog koda dužine 2^{n-1} već samo jednog njegovog elementa. Složenost ove funkcije je zato $O(n)$ (a ne $O(2^n)$ što je složenost ukupne konstrukcije celog Grejovog koda). Zaista, prethodna implementacija zadovoljava jednačinu $T(n) = T(n-1) + O(1)$ i $T(0) = 1$, čije je rešenje $O(n)$.

Funkciju možemo realizovati i iterativno. Primetimo da se Grejov kod formira karakter po karakter sleva na desno, tako da naredba `rez=rez+"0";` ima efekat kao da redom čitamo nule ili jedinice u zavisnosti od toga da li se nalazimo u prvoj ili u drugoj polovini niza.

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

Invarijanta petlje je da je $n_0 \geq n \geq 0$, gde je n_0 početna vrednost broja, 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).

Napomenimo i da se Grejov kôd 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);
}
```

Određivanje prefiksnog obilaska drveta na osnovu infiksnog i postfiksnog obilaska

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 (tada je rešenje jedinstveno). Na primer, razmotrimo naredno drvo.

```
      1
  2       5
3   4   6   7
              8
```

Njegov postfiksni obilazak je `34268751` a infiksni `32416578`. Potrebno je da rekonstruišemo prefiksni obilazak koji je u ovom slučaju `12345678`.

Podsećanja radi navešćemo obilaske u terminima: L-obilazak levog podstabla, k-ispisivanje korena stabla, D-obilazak desnog podstabla. Onda se infiksni obilazak stabla može zapisati kao `L-k-D`, postfiksni obilazak kao `L-D-k` i prefiksni oblik kao `k-L-D`.

Primetimo da se koren drveta onda jednostavno određuje kao poslednji element u postfiksnom obilasku (u našem primeru to je `1`). Tada taj element možemo pronaći i u infiksnom obilasku i na osnovu njegove pozicije odrediti koji elementi pripadaju levom, a koji desnom poddrvetu (u našem primeru se određuje da su `324` elementi infiksnog obilaska levog, a `6578` infiksnog obilaska desnog poddrveta). Na osnovu toga dobijamo i broj elemenata u svakom poddrvetu, pa možemo pročitati i njihove postfiksne obilaske (u primeru će to biti `342` i `6875`). 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).

Na osnovu prethodne diskusije možemo lako napraviti rekurzivnu implementaciju.

```
string nadjiPre(const string& post, const string& in) {
   if (in == "" && post == "")
      return "";

   // citamo koreni element
   char koren = post[post.size() - 1];
   // racunamo duzine levog i desnog podstabla
```

```
   size_t levo = in.find(koren);
   size_t desno = in.size() - levo - 1;

   // u infiksnom zapisu levo podstablo pocinje na poziciji 0
   // desno podstablo pocinje iza korenog elementa
   string in_l   = in.substr(0, levo);
   string in_d   = in.substr(levo + 1, desno);
   // u postfiksnom zapisu levo podstablo pocinje na poziciji 0
   // desno podstablo pocinje odmah iza levog podstabla
   string post_l = post.substr(0, levo);
   string post_d = post.substr(levo, desno);

   // formiramo prefiksan obilazak
   return koren + nadjiPre(post_l, in_l) + nadjiPre(post_d, in_d);
}
```

Kreiranje velikog broja pomoćnih niski možemo jednostavno izbeći tako što ćemo pamtiti početak i kraj postfiksnog, odnosno infiksnog, zapisa koji se trenutno obrađuje. Pored toga pamtićemo i poziciju u prefiksnom zapisu koju trenutno računamo.

```
void nadjiPre(const string& post, int post_od, int post_do,
              const string& in, int in_od, int in_do,
              string& pre, int pre_od) {
  if (post_od > post_do && in_od > in_do)
    return;

  char koren = post[post_do];
  size_t levo = in.find(koren, in_od) - in_od;
  size_t desno = (in_do - in_od) - levo;
  pre[pre_od] = koren;
  nadjiPre(post, post_od, post_od + levo - 1,
        in, in_od, in_od + levo - 1,
        pre, pre_od + 1);
  nadjiPre(post, post_od + levo, post_od + levo + desno - 1,
        in, in_od + levo + 1, in_od + levo + desno,
        pre, pre_od + 1 + levo);
}

string nadjiPre(const string& post, const string& in) {
   string pre(post.size(), ' ');
   nadjiPre(post, 0, post.size() - 1,
            in, 0, in.size() - 1,
            pre, 0);
   return pre;
}
```

Recimo i to da bi algoritam ispravno rekonstruisao i stablo u kom neki čvorovi imaju isti sadržaj, ali u tom slučaju rešenje ne bi bilo jedinstveno (u zavisnosti od

toga koje pojavljivanje korena u infiksnom obilasku odaberemo, dobijali bismo različita ispravna rešenja). Pokušajte da obrazložite zašto je to tako.

Izvođenje iterativnih algoritama induktivno-rekurzivnim pristupom

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 (složenost bi bila $O(kn)$). Moguće je napraviti rešenje složenosti $O(n)$. Primetimo prvo 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.

Ako bi niz imao $2k$ elemenata, rotacija za k mesta bi se vršila tako što bi se razmenili blokovi elemenata koji čine prvu i drugu polovinu niza. U opštem slučaju situacija je malo komplikovanija, ali ponovo svodi na razmenu blokova elemenata. 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 su 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.

Dakle, zadatak se može rešiti induktivno-rekurzivnom konstrukcijom. Prvi slučaj možemo podvesti pod drugi (ili treći) jer se kompletan levi blok zamen-

juje sa desnim, nakon čega ostaje da se razmene prazni blokovi, što ne proizvodi nikakav efekat, pa izlaz iz rekurzije može biti slučaj kada je bilo koji od blokova prazan.

Razmotrimo sledeći primer. Želimo da rotiramo sledeći niz za 4 mesta ulevo.

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

Nakon rotacije trebalo bi da dobijemo

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

Na osnovu opisanog algoritma, rotacija u prethodnom primeru se vrši tako što razmenjujemo levi blok `1 2 3 4` sa desnim blokom `5 6 7 8 9 10`. Pošto je levi blok kraći od desnog bloka, možemo zameniti ceo blok `1 2 3 4` sa prva 4 elementa bloka `5 6 7 8 9 10`.

Tako dobijamo

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

Na taj načini brojevi `5 6 7 8` su došli na svoje mesto, a da bi se od ovog niza dobio krajnji rezultat, potrebno je u delu niza iza njih razmeniti blok `1 2 3 4` i blok `9 10`. Kako je levi blok duži od desnog bloka, prvo na početak dovodimo desni blok i dobijamo

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

Do konačnog rešenja sada možemo doći tako što razmenimo blokove `3 4` i `1 2`. U ovom slučaju je dužina oba bloka jednaka, pa se nakon njihove razmene procedura zaustavlja.

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

Na osnovu ovog razmatranja nameće se rekurzivni algoritam.

```
// funkcija koja razmenjuje blokove koji pocinju na pozicijama p1 i p2
// koji su iste duzine d1
void razmeniManji(int a[], int p1, int p2, int m){
  for(int i=0; i<m; i++){
    int t = a[i+p1];
    a[i+p1] = a[i+p2];
    a[i+p2] = t;
  }
}

// funkcija razmenjuje blok koji pocinje na poziciji p1 i duzine je d1
// i blok koji pocinje na poziciji p2 i duzine je d2
void razmeniBlokove(int a[], int p1, int d1, int p2, int d2) {
  // ako je neki blok prazan nema potrebe za razmenom
  if (d1 == 0 || d2 == 0)
     return;
  if (d1 <= d2) {
```

```
    // razmenjujemo kompletan levi blok sa pocetkom desnog
    razmeniManji(a, p1, p2, d1);
    // iz situacije L.D1.D2 dosli smo u situaciju D1.L.D2 i
    // da bismo dosli u zeljenu situaciju D1.D2.L moramo
    // da razmenimo L i D2
    razmeniBlokove(a, p1 + d1, d1, p2 + d1, d2 - d1);
  } else {
    // razmenjujemo kompletan desni blok sa pocetkom levog
    razmeniManji(a, p1, p2, d2);
    // iz situacije L1.L2.D dosli smo u situaciju D.L2.L1 i
    // da bismo dosli u zeljenu situaciju D.L1.L2 moramo
    // da razmenimo L1 i L2
    razmeniBlokove(a, p1 + d2, d1 - d2, p2, d2);
  }
}

void rotiraj(int a[], int n, int k) {
  // razmenjujemo pocetni blok duzine k i ostatak niza
  razmeniBlokove(a, 0, k, k, n - k);
}
```

U implementaciji smo koristili funkciju `razmeniManji(a, p1, p2, m)` koja razmenjuje blokove duzine `m` koji počinju na pozicijama `p1` i `p2` u nizu `a` (i implementira se trivijalno pomoću jedne petlje i jedne pomoćne promenljive), a može se implementirati i uz pomoć funkcije `swap`. Njena složenost je $O(m)$.

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

Centralna mera progresa u algoritmu je ukupna dužina blokova koji se razmenjuju. Ukupna dužina 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 (novi zbir je $d_1 + (d_2 - d_1) = d_2$, a polazni $d_1 + d_2$). 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 (novi zbir je $(d_1 - d_2) + d_2 = d_1$, a polazni $d_1 + d_2$). Dakle, rekurentna jednačina je u oba slučaja jednaka $T(n) = T(n - d) + O(d)$ i njeno rešenje je $T(n) = O(n)$.

Još jednostavnije, u svakom koraku petlje unutar funkcije `razmeniManji` se tačno jedan element dovodi na svoje finalno mesto, odakle se više ne mrda, a u rekurzivnoj funkciji se osim toga (i rekurzivnih poziva) ne vrši nikakav drugi posao. Pošto se po završetku rekurzije svaki element nalazi na svom mestu izvšeno je tačno n razmena.

Ovaj algoritam možemo izraziti veoma jednostavno i iterativno (rekurzija je

repna, pa se može lako eliminisati).

```
// rotira niz a duzine n za k mesta ulevo
void rotiraj(int a[], int n, int k) {
  k %= n;
  int p1 = 0, d1 = k;
  int p2 = k, d2 = n - k;
  // razmenjujemo blokove [p1, p1 + d1) i [p2, p2 + d2)
  // razmenu vrsimo dok neki od blokova ne postane prazan
  while (d1 != 0 && d2 != 0) {
     if (d1 <= d2) {
        // razmenjujemo kompletan levi blok sa pocetkom desnog
        razmeniManji(a, p1, p2, d1);
        // iz situacije L.D1.D2 dosli smo u situaciju D1.L.D2 i
        // da bismo dosli u zeljenu situaciju D1.D2.L moramo
        // da razmenimo L i D2
        p1 += d1; p2 += d1;
        d2 -= d1;
     } else {
        // razmenjujemo kompletan desni blok sa pocetkom levog
        razmeniManji(a, p1, p2, d2);
        // iz situacije L1.L2.D dosli smo u situaciju D.L2.L1 i
        // da bismo dosli u zeljenu situaciju D.L1.L2 moramo
        // da razmenimo L1 i L2
        p1 += d2;
        d1 -= d2;
     }
  }
}
```

Na kraju, mozemo ukloniti i pomoćnu funkciju `razmeniManji` i dobiti veoma elegantan i efikasan algoritam.

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

// maksimalni broj elemenata niza odredjen tekstom zadatka
const int N_MAX = 100000;

int main() {
  // ucitavanje podataka
  int k, n;
  cin >> n >> k;
  int a[N_MAX];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ciklicko pomeranje za k mesta je isto sto i ciklicko pomeranje za
  // k % n mesta
```

```
  k %= n;

  // vrsimo rotiranje niza razmenom blokova
  int l = 0; // pocetak levog bloka
  int d = k; // pocetak desnog bloka
  // razmenjujemo blokove L i D tj. blokove [l, k) i [d, n)
  // razmenu vrsimo dok neki od blokova ne postane prazan
  while (l != k && d != n) {
    // menjamo tekuce elemente u blokovima
    swap(a[l++], a[d++]);

    if (l == k) // stigli smo do kraja levog bloka (levi blok je kraci)
      // razmenili smo levi blok sa pocetkom desnog i iz L.D1.D2
      // dosli u situaciju D1.L.D2 i jos treba da razmenimo blokove
      // L i D2, a to su blokovi [l, d) i [d, n),
      // tako da k treba postaviti na d
      k = d;

    if (d == n) // stigli smo do kraja desnog bloka (desni blok je kraci)
      // razmenili smo pocetak levog bloka sa desnim i iz L1.L2.D
      // dosli u situaciju D.L2.L1 i jos treba da razmenimo blokove
      // L2 i L1, a to su blokovi [l, k) i [k, n),
      // tako da d treba postaviti na k
      d = k;
  }

  // ispisujemo rotirani niz
  for (int i = 0; i < n; i++)
    cout << a[i] << " ";
  cout << endl;

  return 0;
}
```

Bez komentara, centralni deo prethodnog koda izgleda ovako.

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

Veoma interesantno bi bilo dokazati direktno da je ovaj algoritam korektan. Za početak ni njegovo zaustavljanje nije trivijalno. Dokažimo da se prethodni algoritam zaustavlja. Jedna od invarijanti prethodne petlje je da je $0 \leq l \leq$

$k \leq d \leq n$. Dokažimo to. Pošto je $0 < n$, $0 \leq k$ i $k < n$, na početku važi $0 = l \leq k = d < n$, pa je invarijanta ispunjena.

Na osnovu uslova petlje, kada se uđe u telo petlje znamo da je $0 \leq l < k \leq d < n$. Nakon toga se l i d uvećavaju za 1, pa važi $0 \leq l \leq k \leq d \leq n$. Ako je $l = k$, tada se k postavlja na d, pa odnos i dalje ostaje da važi. Slično, ako je $d = n$, tada se d postavlja na k, pa odnos i dalje ostaje da važi. Na osnovu ovoga znamo da ova invarijanta ostaje ispunjena i nakon izvršavanja tela petlje. Potrebno je još dokazati da se u nekom trenutku mora dogoditi da je $l = k$ ili da je $d = n$. Međutim, u svakom koraku petlje promenljiva l se uvećava za 1, dok se gornja granica n ne menja. Stoga znamo da će se nakon najviše n koraka desiti da je $l = n$. Na osnovu invarijante tada će morati da važi da je $l = k = d = n$, pa će se petlja zaustaviti. Ujedno vidimo i da se u najgorem slučaju petlja izvršava n puta, pa je algoritam složenosti $O(n)$.

Duplikati u nizu od 0 do $n-1$

Problem: Dat je niz od n brojeva u kom svi elementi pripadaju intervalu od 0 do $n-1$. Neki elementi iz ovog intervala možda nedostaju, a neki se možda ponavljaju. Napisati program koji ispisuje sve one koji nedostaju i sve one koji se ponavljaju (u proizvoljnom redosledu).

Direktno rešenje bi linearnom pretragom za svaki broj proverilo da li se pojavilo 0, 1 ili više puta i na osnovu toga odredilo rezultat. Složenost takvog rešenja bi bila $O(n^2)$, jer bi se svaki nedostajući element tražio u n koraka, a njih može biti $n-1$.

Mnogo bolje rešenje se dobija ako se niz sortira. U sortiranom nizu duplikati se nalaze jedan do drugog, a nedostajuće elemente bismo pronašli između onih elemenata niza čija je razlika veća od 1. Složenošću bi dominiralo sortiranje, koje bi se moglo obaviti u $O(n \log n)$, dok bi druga faza zatim mogla biti urađena u $O(n)$.

Međutim, možemo bolje od toga. Zahvaljujući veoma specifičnom rasponu elemenata niza, sortiranje možemo uraditi posebno prilagođenim algoritmom u vremenu $O(n)$. Pokušavamo da niz uredimo tako da ako se vrednost i javlja u nizu, da bar jedno njeno pojavljivanje bude upravo na poziciji i (tj. da je $a[i] = i$). Kada se niz uredi tako, onda u jednom prolazu možemo odrediti i duplikate i nedostajuće elemente. Ako postoji neka vrednost j koja se javlja na poziciji koja nije njena (ako je $j = a[i]$ i $j \neq i$), tada možemo zaključiti dve stvari. Prvo, pošto se j javlja i na svojoj poziciji j (tj. $a[j] = j$ zbog pretpostavke o uređenju niza), ovo je bar drugo pojavljivanje vrednosti j (na pozicijama i i j), pa je to duplikat. Drugo, element sa vrednošću i ne postoji u nizu, jer da postoji, morao bi se javiti upravo na toj poziciji i (koju upravo razmatramo). Ostaje samo još pitanje kako efikasno možemo da uredimo niz na opisani način.

Ključna ideja je da stalno pokušavamo da unapredimo stanje niza tako što

elemente postavljamo na mesto na kom treba da budu. Krećemo od početka niza i pokušavamo tekući element (koji je na tekućem mestu) da dovedemo na njegovo mesto. Dve situacije su moguće. Ako se na mestu na kom treba da bude tekući element nalazi upravo taj element, onda je on ili već na svom mestu ili smo našli duplikat. U toj situaciji smo završili sa obradom tekućeg elementa i možemo preći na sledeći. Ako se na mestu na kom treba da bude tekući element nalazi neki drugi element, onda se oni mogu razmeniti, čime se povećava broj elemenata koji su na svom mestu. Pošto element koji je tom razmenom došao na tekuću poziciju možda nismo još analizirali, tekuću poziciju ne menjamo.

```
void sortiraj(int a[], int n) {
  int i = 0;
  while (i < n) {
    // citamo element a[i] i proveravamo da li se nalazi
    // na svojoj poziciji a[a[i]]
    if (a[a[i]] == a[i])
      // ako se nalazi prelazimo na naredni element
      i++;
    else
      // ako se ne nalazi, dovodimo ga na svoju poziciju
      // a element sa kojim smo ga zamenili obradjujemo u
      // narednom prolazu petlje
      swap(a[i], a[a[i]]);
  }
}
```

Pokažimo prvo kako ovaj algoritam radi na jednom primeru.

```
0 1 2 3 4 5 6 7 8
-----------------
.8 1 6 5 0 5 2 3 1      8 dovodimo na njegovo mesto
.1 1 6 5 0 5 2 3 8      1 je duplikat
1 .1 6 5 0 5 2 3 8      1 je na svom mestu
1 1 .6 5 0 5 2 3 8      6 dovodimo na njegovo mesto
1 1 .2 5 0 5 6 3 8      2 je na svom mestu
1 1 2 .5 0 5 6 3 8      5 je duplikat
1 1 2 5 .0 5 6 3 8      0 dovodimo na njegovo mesto
0 1 2 5 .1 5 6 3 8      1 je duplikat
0 1 2 5 1 .5 6 3 8      5 je na svom mestu
0 1 2 5 1 5 .6 3 8      6 je na svom mestu
0 1 2 5 1 5 6 .3 8      3 dovodimo na njegovo mesto
0 1 2 3 1 5 6 .5 8      5 je duplikat
0 1 2 3 1 5 6 5 .8      8 je na svom mestu
```

Dokažimo korektnost ovog algoritma.

Lema: Važi da je $0 \leq i \leq n$ i da su svi elementi u intervalu $[0, i)$ takvi da je $a_{a_i} = a_i$.

Pretpostavimo da tvrđenje važi na ulasku u petlju. Tada je $i < n$. Ako je $a_{a_i} = a_i$ važi da se niz ne menja i da je $i' = i + 1$. Jasno je da se u tom slučaju invarijanta održava (jer je važila za sve elemente pre pozicije i i važi i za element na poziciji i). Ako je $a_{a_i} \neq a_i$ tada se vrši razmena elemenata na pozicijama i i a_i. To znači da se nakon razmene na poziciji a_i nalazi element sa vrednošću a_i. Ako je $a_i \geq i$, tada se razmenom ne utiče na elemente na pozicijama $[0, i)$ i invarijanta trivijalno ostaje da važi. Ako je $a_i < i$, tada se niz menja, međutim, i u novom nizu će na poziciji a_i biti element koji je jednak a_i, pa će invarijanta ostati da važi.

Teorema: Po završetku petlje, za svaki element $0 \leq i < n$ važi da ako je $a_i \neq i$, tada je a_i duplikat, a vrednost i se ne javlja u nizu.

Po završetku petlje za svaki element niza važi da je $a_{a_i} = a_i$. Naime, na osnovu invarijante je $0 \leq i \leq n$, a pošto je petlja završena važi da je $i \geq n$, pa je $i = n$. Ponovo na osnovu invarijante za sve pozicije i iz intervala $[0, n)$, a to su sve pozicije u nizu, znamo da je $a_{a_i} = a_i$. Ako je $a_i \neq i$, znamo da se a_i javlja i na poziciji i i na poziciji a_i, pa je duplikat. Dokažimo da tada i ne može da se javi u nizu. Ako bi za neko i' važilo da je $a_{i'} = i$, tada bi važilo $a_{a_{i'}} = a_{i'}$, jer to važi za svaku poziciju, pa i i'. Uvrštavanjem $a_{i'} = i$, dobijamo da bi moralo da važi $a_i = i$, što je direktna kontradikcija u odnosnu na pretpostavku da je $a_i \neq i$.

Dokaz zaustavljanja i analiza složenosti je možda još interesantnija. Algoritam napreduje tako što se ili i pomera ka desnom kraju niza ili tako što se povećava broj pozicija k takvih da je $a_k = k$. Ako je m broj takvih pozicija, možemo razmatrati broj $i + m$. Dokažimo da svako izvršavanje tela petlje povećava ovaj broj bar za 1. Ako je $a_{a_i} = a_i$, tada se i povećava za 1. U suprotnom, se vrši razmena elemenata na poziciji i i a_i. Na osnovu pretpostavke, na poziciji a_i nije mogao da bude element a_i. Ni na poziciji i nije mogao da bude element a_i, jer da jeste, bilo bi $a_i = i$ i važilo bi da je $a_{a_i} = a_i$, što je opet suprotno u odnosu na pretpostavku. Dakle, pre razmene nijedna od ove dve pozicije nije doprinosila broju m. Nakon razmene, znamo da je sigurno na poziciji a_i element a_i, jer razmenom a_i prebacujemo na poziciju a_i. Ne znamo da li se nakon razmene na poziciji i nalazi a_i, ali to nam nije ni bitno, jer već na osnovu pozicije a_i znamo da je m uvećan bar za jedan (jer se ostali elementi u nizu nisu pomerali tako da oni nastavljaju da doprinose vrednosti m kao i pre razmene). Veličina $i + m$ je odozgo ograničena sa $2n$. Naime znamo da je $i \leq n$, kao i da je broj m odozgo ograničen sa n (jer u nizu dužine n ima ukupno n različitih pozicija). Zbog toga znamo da se telo petlje može izvršiti najviše $2n$ puta, pa je složenost ovog algoritma sortiranja $O(n)$.

Primenimo sada isti ovaj algoritam da rešimo naredno uopštenje problema koji smo već razmatrali.

Problem: Iz niza u koji su bili upisani svi brojevi od 0 do n, izbačeno je k elemenata. Odredi koji su elementi izbačeni.

Zadatak se lako svodi na prethodni tako što se niz dužine $n-k$ proširi do dužine n, popunjavajući ga nekim duplikatima tj. elementima koji se već nalaze u nizu. Najlakše je da na kraj niza dodamo k pojavljivanja elementa a_0 i da zatim primenimo prethodni algoritam za nalaženje nedostajućih elemenata.

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 ako i samo ako osoba i poznaje osobu j).

Direktan način je da za svaku osobu proverimo da li zadovoljava uslov zvezde.

```
bool poznajeNekog(bool poznaje[MAX][MAX], int i) {
    for (int j = 0; j < poznaje[i].size(); j++)
       if (i != j && poznaje[i][j])
          return true;
    return false;
}

bool sviJePoznaju(bool poznaje[MAX][MAX], int i) {
    for (int j = 0; j < poznaje.size(); j++)
       if (i != j && !poznaje[j][i])
          return false;
    return true;
}

int zvezda(bool poznaje[MAX][MAX]) {
   for (int i = 0; i < poznaje.size(); i++)
       if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
          return i;
   return -1;
}
```

Složenost najgoreg slučaja ovog algoritma je $O(n^2)$, mada je fer priznati da se ta složenost veoma teško dostiže, jer je za očekivati da će se obe provere u slučaju kada kandidat nije zvezda prekinuti mnogo pre nego što se stigne do kraja niza.

Poboljšanje možemo pokušati induktivno-rekurzivnim pristupom. Prva ideja je da problem pronalaženja zvezde u skupu od n osoba svedemo na problem pronalaženja zvezde u skupu od $n-1$ osoba.

- Baza indukcije je prazan skup osoba u kome ne postoji zvezda.
- Pretpostavimo kao induktivnu hipotezu da umemo da izračunamo zvezdu u skupu od $n-1$ osoba. Razdvojimo skup od n osoba na podskup od početnih $n-1$ osoba i poslednju osobu o_{n-1}. Zvezda celog skupa može

biti ili zvezda tog podskupa ili osoba o_{n-1} (jer ako je osoba zvezda nekog skupa onda je ona ujedno i zvezda svakog podskupa kojem pripada).

1. Ako u podskupu od $n-1$ osoba postoji zvezda, da bi ona bila zvezda celog skupa potrebno je da je poznaje osoba o_{n-1} i da ona ne poznaje osobu o_{n-1}. To možemo proveriti u vremenu $O(1)$.
2. U suprotnom (ako u podskupu ne postoji zvezda ili ako zvezda postoji, ali ona poznaje osobu o_{n-1} ili osoba o_{n-1} ne poznaje nju) moramo još ispitati da li je osoba o_{n-1} zvezda. To pitanje nikako ne zavisi od toga da li u podskupu postoji zvezda i da bi se to ispitalo potrebno je posebno proveriti da li svih prethodnih $n-1$ osoba poznaje osobu o_{n-1} i da li ona ne poznaje nikoga od njih. Za to je potrebno $O(n)$ koraka.

Dakle, prethodna konstrukcija nam daje korektan induktivni algoritam, međutim, složenost najgoreg slučaja takvog algoritma je $T(n) = T(n-1) + O(n)$, što je, kao što znamo, $O(n^2)$.

```
int zvezda(const vector<vector<bool>>& poznaje, int n) {
  if (n == 0)
    return -1;
  int z = zvezda(poznaje, n-1);
  if (z != -1 && poznaje[n-1][z] && !poznaje[z][n-1])
    return z;
  if (!poznajeNekog(poznaje, n-1) && sviJePoznaju(poznaje, n-1))
    return n-1;
  return -1;
}

int zvezda(const vector<vector<bool>>& poznaje) {
  return zvezda(poznaje, poznaje.size());
}
```

Rekurzija se veoma jednostavno uklanja, ali složenost i dalje ostaje nepovoljna.

```
int zvezda(const vector<vector<bool>>& poznaje) {
  int z = -1;
  for (int i = 0; i < poznaje.size(); i++) {
     if (z == -1 || !poznaje[i][z] || poznaje[z][i])
       if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
         z = i;
       else
         z = -1;
  }
  return z;
}
```

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

efikasne induktivne konstrukcije je da veoma brzo i jednostavno (samo jednim pitanjem) 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 (jer su sve ostale osobe eliminisane na osnovu toga što smo utvrdili da ne mogu biti zvezde). Za tu jedinu preostalu osobu onda možemo direktno ispitati da li je zvezda ili nije. Eliminacija ne-zvezde iz skupa se može izvršiti krajnje 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 (jer zvezda nikoga ne poznaje). Ako je odgovor odričan, onda osoba B nije zvezda (jer zvezdu svi poznaju). Algoritam zasnovan na ovom postupku zadovoljava jednačinu $T(n) = O(1) + T(n-1)$, čije je rešenje $O(n)$. Posle ovoga, 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 obe faze $O(n)$, čime se dobija algoritam čiji je najgori slučaj mnogo efikasniji od prethodnog.

Ostaje pitanje tehničke realizacije, odnosno pitanje kako da čuvamo skup kandidata koji nisu još eliminisani i kako da iz njega biramo dve osobe koje ćemo porediti. Jedna mogućnost bi bila da su sve osobe složene na jedan stek (o njima ste već govorili u kursevima programiranja, a još detaljnije ćemo govoriti kasnije). Poredimo dve osobe sa vrha steka i na stek vraćamo samo onu koja nije eliminisana njihovim poređenjem. Postupak nastavljamo dok stek ne postane jednočlan.

Ipak, biće prikazano još jednostavnije rešenje koje koristi dva pokazivača. Prvi pokazivač, i, pokazuje na osobu koja je trenutni kandidat za zvezdu, tj. na prvu osobu u nizu za koju još nije ustanovljeno da nije zvezda. Drugi pokazivač, j, pokazuje na osobu za koju se proverava da li je tekući kandidat za zvezdu poznaje. Ukoliko osoba i ne poznaje osobu j, i je i dalje kandidat, a j sigurno nije zvezda (jer svi moraju da poznaju zvezdu), pa se j pomera na sledeću osobu. Na početku su `i` i `j` na susednim pozicijama ($i = 0$, $j = 1$) i osobe se obrađuju sleva na desno sekvencijalno pa važi da osobe strogo između i i j nisu zvezde. Ukoliko osoba i poznaje osobu j, onda i nije zvezda (jer zvezda ne poznaje nikoga), ali j možda jeste, pa, pošto između i i j niko nije zvezda, prvi sledeći kandidat je tekuća osoba j i i se pomera na j, a j na prvu sledeću osobu.

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

Lema: Dokažimo da je invarijanta prethodne petlje da nijedan element u intervalu $[0, i)$ i nijedan element u intervalu (i, j) ne može biti zvezda (pri čemu je $0 \leq i < j \leq n$).

- Na početku je $i = 0$ i $j = 1$, pa su oba intervala prazna, a uslov važi (pod pretpostavkom da je $n > 0$).
- Pretpostavimo da uslov važi pri ulasku u petlju. Ako osoba i poznaje osobu j tada ona ne može biti zvezda. Pošto na osnovu pretpostavke znamo da zvezde ne mogu biti ni osobe $[0, i)$ kao ni osobe (i, j) i kako je $i' = j$, znamo da nakon tela petlje zvezde sigurno nisu ni osobe u intervalu $[0, i')$. Pošto je $j' = j + 1$, u tom slučaju je interval (i', j') prazan, pa trivijalno zadovoljava uslov. Ako osoba i ne poznaje osobu j tada znamo da j ne može da bude zvezda. Tada je $i' = i$, a $j' = j + 1$, pa na osnovu pretpostavke znamo da zvezde ne mogu biti osobe iz intervala $[0, i')$, znamo i da ne mogu biti osobe iz intervala (i', j), a pošto ni j ne može biti zvezda, znamo da zvezde ne mogu biti osobe iz intervala (i', j'). Odnos između promenljivih trivijalno ostaje očuvan u oba slučaja.

Teorema: Funkcija ili ispravno pronalazi zvezdu ili vraća -1 ako zvezda ne postoji.

Kada se petlja završi nije $j < n$, pa je na osnovu invarijante $j = n$. Tada znamo da zvezde ne mogu biti osobe iz intervala $[0, i)$ kao ni osobe iz intervala (i, n). Dakle, jedini kandidat za zvezdu je osoba i. Za nju se eksplicitno proverava traženi uslov, tako da funkcija vraća korektnu vrednost.

Naglasimo još da je vreme učitavanja matrice $O(n^2)$, čime se, nažalost, poništava dobitak na efikasnosti ovog algoritma, pa je vreme realnog izvršavanja programa uključujući fazu učitavanja praktično nepromenjeno ovom divnom optimizacijom.

Apsolutni pobednik na glasanju

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.

Postoji mnogo načina da se ovaj zadatak reši. Ako imamo na raspolaganju efikasne strukture podataka, možemo prebrojati glasove svih kandidata, odrediti onog sa najvećim brojem glasova i videti da li je osvojio strogo više od polovine ukupnog broja glasova. Možemo odrediti medijanu niza glasova i tako utvrditi kandidata za apsolutnog pobednika (jer ako postoji apsolutni pobednik, on se sigurno nalazi na središnjoj poziciji tj. središnje dve pozicije u nizu, po kom god kriterijumu sortirali glasove, dok god su nam glasovi za istog kandidata uzastopni). O efikasnim algoritmima za određivanje medijane (bez sortiranja celog niza) biće više reči kasnije.

Ipak, u ovom poglavlju prikazaćemo sasvim elementaran i izrazito elegantan algoritam za rešenje ovog problema koji su uveli Bojer i Mur u radu *"A fast majority vote algorithm"* (interesantno, u originalnom radu algoritam je uveden u cilju prikaza mogućnosti automatske formalne verifikacije softvera).

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).

Rešenje se zasniva na tome da ako imamo apsolutnog pobednika u većem skupu glasova, i izbacimo bilo koja dva različita glasa, onda će manji skup imati apsolutnog pobednika samo ako ga je imao i veći skup, i u pitanju je isti apsolutni pobednik. Dokažimo ovo. 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 $m > n/2 > (n-2)/2$ pa apsolutni pobednik nije promenjen. 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 $m-1 > (n-2)/2 = n/2-1$, jer je $m > n/2$ i apsolutni pobednik nije promenjen. Dakle, izbacivanjem dva različita glasa, redukujemo dimenziju problema. 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.

Obratimo pažnju na to da postojanje apsolutnog pobednika u manjem skupu ne implicira to da u većem skupu postoji apsolutni pobednik. Preciznije, pretpostavimo da je p_1 apsolutni pobednik skupa od $n-2$ elementa i da on ima m_1 glasova. Tada je $m_1 > (n-2)/2$, tj. $m_1 > n/2 - 1$. Što znači da je on apsolutni pobednik i u situaciji kada je $m_1 = n/2$. Razmotrimo šta se dešava kada se veći skup (od n elemenata) dobija dodavanjem dva različita glasa od kojih ni jedan nije glasao za p_1. U tom većem skupu, broj glasova za p_1 ostaje m_1 ali nejednakost $m_1 > n/2$ nije ispunjena pa p_1 nije apsolutni pobednik u tako formiranom skupu od n elemenata.

Opisali smo, dakle, induktivno-rekurzivnu konstrukciju koja nam može omogućiti da dobijemo kandidata za apsolutnog pobednika (što će se desiti ako skup glasova izbacivanjem parova različitih glasova svedemo na neprazan skup u kom su svi glasovi za istog kandidata) ili da utvrdimo da apsolutni pobednik ne postoji (što će se desiti ako skup glasova izbacivanjem parova različitih glasova svedemo na prazan skup). Naglasimo i da je apsolutni pobednik jedinstven, pa nije moguće da postoje dva različita kandidata za apsolutne pobednike.

Predstavićemo jednu efikasnu implementaciju. Pretpostavimo da se skup svih glasova može podeliti na dva disjunktna podskupa. Svi glasovi u prvom skupu su za istu (proizvoljnu) osobu, dok se u drugom skupu nalaze parovi glasova za

različite osobe (ovo će biti centralna invarijanta petlje). Izbacivanjem jednog po jednog para glasova iz drugog skupa, problem se može svesti na sve manju i manju dimenziju, sve dok ne ostanu samo elementi prvog skupa. Ako je prvi skup prazan, možemo zaključiti da ne postoji apsolutni pobednik u početnom skupu svih glasova, a ako nije, onda je osoba za koju su svi glasovi u prvom skupu jedini kandidat za apsolutnog pobednika.

Ostaje pitanje kako da napravimo neku podelu skupa svih glasačkih listića na ova dva podskupa. Krenućemo od toga da su oba podskupa prazna i polako ćemo određivati njihov sadržaj obrađujući jedan po jedan glas. Primetimo da je dovoljno 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 iz prvog skupa, tekući glas možemo dodati u prvi skup (u drugom skupu će se i dalje nalaziti parovi različitih glasova, a svi glasovi u prvom skupu će i dalje biti za istu osobu) i možemo uvećati tekući broj glasova za jedan. 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 (u prvom skupu će i dalje svi glasovi biti za istu osobu, a u drugom će biti svi raniji parovi različitih glasova i ovaj novododati par za koji smo takođe sigurni da je par glasova za različite osobe) i možemo smanjiti tekući broj glasova za jedan.

Primetimo i da nemamo garancije da će poslednji kandidat za apsolutnog pobednika biti stvarno apsolutni pobednik, tako da se ta provera mora izvršiti na kraju programa.

```
bool apsolutniPobednik(int glasovi[], int n, int& pobednik) {
  // odredjujemo kandidata za pobednika (ako postoji) i broj glasova
  // koji preostane kada se poniste parovi razlicitih glasova
  int kandidat;
  int glasovaZaKandidata = 0;
  for (int i = 0; i < n; i++)
    // ako nema kandidata ili ako je glas za tekuceg kandidata
    // broj glasova uvecavamo za 1
    if (glasovaZaKandidata == 0 || glasovi[i] == kandidat) {
      kandidat = glasovi[i];
      glasovaZaKandidata++;
    } else
      // ako je glas za nekog drugog smanjujemo broj glasova za kandidata
      glasovaZaKandidata--;

  // proveravamo da li postoji kandidat za pobednika i da li je
  // ostvario vise od n/2 glasova
  if (glasovaZaKandidata > 0 &&
      count(glasovi, next(glasovi, n), kandidat) > n / 2) {
    pobednik = kandidat;
    return true;
  } else
    return false;
```

```
}
```

Složenost i faze određivanja i faze provere kandidata je očigledno linearna, pa je ceo algoritam složenosti $O(n)$. Osim za čuvanje niza glasova, ne angažuje se nikakva značajna dodatna memorija.

Ojačavanje induktivne hipoteze

U nekim situacijama se dobitak na efikasnosti može postići time što pretpostavimo da je induktivna hipoteza jača tj. da se u svakom koraku umesot jedne izračunava nekoliko različitih vrednosti. Razmotrimo nekoliko takvih primera.

Izračunavanje vrednosti polinoma

Problem: Dat je niz realnih brojeva $a_0, a_1, \dots, a_n$ 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 $P_n(x) = 0$ (jer je suma $\sum_{i=0}^{n} a_i x^i$ prazna, pa joj je zbir po definiciji 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. Postoje razni načini da se to efikasno uradi, ali to svakako nije trivijalan problem. Ako bismo vrednost stepena izračunavali naivno (množenjem sa x n puta), dobili bismo ovakav kôd.

```
int stepen(int x, int n) {
   if (n == 0)
      return 1;
   return stepen(x, n-1) * x;
}

int vrednostPolinoma(int a[], int n, int x) {
   if (n == 0)
      return a[0];
   return a[n] * stepen(x, n) + vrednostPolinoma(a, n-1, x);
}
```

Pod pretpostavkom da funkcija `stepen` ispravno vrši stepenovanje, dokažimo da je naša rekurzivna definicija dobra.

Teorema: Funcija `vrednost_polinoma` za dati niz a čija je dužina veća ili jednaka n, izračunava vrednost $P_n(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + \dots a_1 x + a_0$.

Tvrđenje dokazujemo indukcijom po n. Kao bazu možemo uzeti slučaj $n = 0$ kada funkcija ispravno izračunava zbir $P_0(x) = \sum_{i=0}^{0} a_i x^i = a_0$. Kao induktivnu hipotezu pretpostavimo da rekurzivni poziv za vrednost $n-1$ ispravno izračunava $P_{n-1}(x) = \sum_{i=0}^{n-1} a_i x^i = a_{n-1}x^{n-1} + \ldots a_1 x + a_0$ i da poziv funkcije `stepen` ispravno izračunava x^n. Funkcija vraća zbir vrednosti $a_n \cdot x^n$ i vrednosti rekurzivnog poziva za koji na osnovu induktivne hipoteze znamo da je jednak $P_{n-1}(x) = a_{n-1}x^{n-1} + \ldots a_1 x + a_0$. Zato je povratna vrednost jednaka $P_n(x) = a_n x^n + \ldots a_1 x + a_0$, što je i trebalo dokazati.

Naravno, mnogo prirodnije za imperativne jezike je osloboditi se rekurzije, što je u ovom slučaju prilično jednostavno.

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

Broj množenja u ovom kodu je prilično očigledno $O(n^2)$.

Efikasnost možemo značajno popraviti ako 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}.

Dve povratne vrednosti možemo realizovati preko para ili preko prenosa po referenci (ili preko pokazivača).

```
pair<int, int> vrednostPolinoma(int a[], int n, int x) {
   if (n == 0)
      return make_pair(a[0], x);
   int v, s;
   tie(v, s) = vrednostPolinoma(a, n-1, x);
   return make_pair(a[n] * s + v, s * x);
}

void vrednostPolinoma(int a[], int n, int x, int& v, int& s) {
   if (n == 0) {
      v = a[0]; s = x;
      return;
   }
   vrednostPolinoma(a, n-1, x, v, s);
   v = a[n] * s + v;
   s = s * x;
}
```

Dokažimo korektnost ove implementacije.

Teorema: Funcija `vrednost_polinoma` za dati niz a čija je dužina veća ili jednaka n, izračunava par koji čini vrednost polinoma $P_n(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + ... a_1 x + a_0$ i vrednost x^{n+1}.

Bazu indukcije čini slučaj $n = 0$ i tada funkcija ispravno izračunava vrednost $\sum_{i=0}^{0} a_i x^i = a_0$ i $x^{0+1} = x$. Pretpostavimo da rekurzivni poziv ispravno izračunava par vrednosti $v = \sum_{i=0}^{n-1} a_i x^i$ i $s = x^{(n-1)+1} = x^n$. Funkcija tada uspešno izračunava par vrednosti $v' = a_n \cdot s + v$, što je na osnovu induktivne hipoteze jednako $a_n x^n + \sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{n} a_i x^i$ i vrednosti $s' = s \cdot x$, što je na osnovu induktivne hipoteze jednako $x^n \cdot x = x^{n+1}$, čime je tvrđenje dokazano.

Ojačavanje induktivne hipoteze se u ovom slučaju ogledalo u tome da umesto jedne vrednosti, funkcija izračunava i vraća par vrednosti. Vrednost stepena je korišćenja da bi se izračunala nova vrednost polinoma i jako je zgodno bilo što smo i nju dobili direktno iz rekurzivnog poziva. To što smo dobili, moramo da nadoknadimo time što se nakon rekurzivnog poziva moramo malo više potruditi da izračunamo i vratimo ne samo vrednost polinoma koja nas zanima, nego i vrednost stepena, koja će nam biti potrebna na narednom nivou rekurzije.

Osnovna teorema indukcije glasi

$$A(0) \wedge (\forall n)(A(n) \Rightarrow A(n+1)) \Rightarrow (\forall n)(A(n)).$$

Ojačavanje induktivne hipoteze se dakle svodi na to da umesto hipoteze $A(n)$ pretpostavimo ojačanu hipotezu oblika $A(n) \wedge B(n)$. Teorema indukcije u ovom slučaju glasi

$$(A(0) \wedge B(0)) \wedge ((\forall n)(A(n) \wedge B(n) \Rightarrow A(n+1) \wedge B(n+1)) \Rightarrow (\forall n)(A(n)).$$

Prethodno rekurzivno rešenje vraća uređen par brojeva i direktno odgovara pomenutoj induktivno-rekurzivnoj konstrukciji, ali je prilično neprirodno za imperativni jezik. Mnogo prirodnije je osloboditi se rekurzije. Iterativni kôd se može napisati na sledeći način.

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

Dokažimo korektnost ove iterativne implementacije korišćenjem invarijante petlje.

Lema: Invarijanta petlje je da promenljiva `v` sadrži vrednost $P_{i-1}(x)$, a promenljiva `s` vrednost x^i, kao i da je $0 \leq i \leq n+1$.

Pre petlje je $i = 0$, $s = 1 = x^0$ a $v = 0$ (što je vrednost $P_{-1}(x)$). Efekat tela petlje je to da je $v' = v + a_i \cdot s$, da je $s' = s \cdot x$ i da je $i' = i + 1$, pa je, na osnovu pretpostavke $v' = a_i \cdot x^i + P_{i-1}(x) = P_i(x) = P_{i'-1}(x)$ i da je $s' = x^i \cdot x = x^{i+1} = x^{i'}$.

Teorema: Nakon petlje promenljiva `v` sadrži vrednost $P_n(x)$.

Na osnovu invarijante znamo da na kraju petlje važi $i = n + 1$ (jer nije $i \leq n$, a važi $0 \leq i \leq n + 1$). Opet na osnovu invarijante znamo da promenljiva `v` sadrži $P_{i-1}(x)$, što je jednako $P_n(x)$, jer je $i = n + 1$.

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.

Na kraju, recimo i da postoji drugačija induktivno-rekurzivna konstrukcija koja pomaže da se problem reši i jednostavnije i efikasnije (u ovoj varijanti nije nam potrebno ojačavanje induktivne hipoteze). Ključna ideja je da izračunavanje vršimo sdesna nalevo, umesto sa leva na desno. Naime, polinom $P_n(x)$ možemo zapisati kao $(a_n x^{n-1} + a_{n-1} x^{n-2} + a_1) \cdot x + a_0$. 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*.

```
int vrednostPolinoma(int a[], int i, int n, int x) {
   if (i > n)
      return 0;
   return vrednostPolinoma(a, i+1, n, x)*x + a[i];
}

int vrednostPolinoma(int a[], int n, int x) {
   return vrednostPolinoma(a, 0, n, x);
}
```

Oslobađanje od rekurzije daje mnogo prirodniju formulaciju.

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

Dokažimo korektnost ove implementacije korišćenjem invarijante petlje.

Lema: Invarijanta petlje je da je za $-1 \leq i \leq n$ vrednost promenljive `v` jednaka $\sum_{k=i+1}^{n} a_k x^{k-i-1}$.

U početku je $i = n$, a $v = 0$, pa tvrđenje važi jer je suma prazna, pa joj je zbir po definiciji nula. Efekat tela petlje je da je $v' = v \cdot x + a_i$, pa je, na osnovu pretpostavke, $v' = \left(\sum_{k=i+1}^{n} a_k x^{k-i-1}\right) \cdot x + a_i = \sum_{k=i+1}^{n} a_k x^{k-i} + a_i = \sum_{k=i}^{n} a_k x^{k-i-1}$. Pošto je $i' = i - 1$ i pošto je $i \geq 0$, invarijanta ostaje očuvana.

Teorema: Nakon petlje promenljiva `v` sadrži vrednost $P_n(x)$.

Pošto uslov petlje nije ispunjen znamo da je nakon petlje $i < 0$. Na osnovu invarijante znamo da je $-1 \leq i \leq n$, pa zato nakon petlje mora da važi $i = -1$. Na osnovu invarijante promenljiva v sadrži vrednost, $\sum_{k=i+1}^{n} a_k x^{k-i-1}$, što je za $i = -1$ jednako $\sum_{k=0}^{n} a_k x^k$, što je tražena vrednost polinoma $P_n(x)$.

I u ovom algoritmu je broj množenja $O(n)$, međutim, ovde imamo jedno množenje i jedno sabiranje po koraku tela petlje, a u prethodnom smo imali dva množenja i jedno sabiranje, pa je ova implementacija za konstantni faktor brža.

Faktor 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;
};
```

Prazno stablo predstavlja se specijalnim pokazivačem `nullptr`.

Direktna induktivno-rekurzivna konstrukcija u ovom primeru ne daje rezultate jer faktor ravnoteže čvora ne zavisi od faktora ravnoteže levog i desnog poddrveta (tj. njihovih korenih čvorova), već od visine tih drveta. Sa druge strane, visinu drveta veoma jednostavno možemo izračunati na osnovu induktivno-rekurzivne konstrucije.

```
int visina(cvor* koren) {
  // visina praznog stabla je nula
  if (koren == nullptr)
     return 0;
  // visina untrašnjeg čvora
  return max(visina(koren->levo), visina(koren->desno)) + 1;
}
```

Ako pretpostavimo da se levo i desno nalazi isti broj elemenata ova funkcija zadovoljava jednačinu $T(n) = 2 \cdot T(n/2) + O(1)$, čije je rešenje na osnovu master teoreme $O(n)$. Složenost je takva i bez te pretpostavke (ako je drvo degenerisano u listu, složenost zadovoljava jednačinu $T(n) = T(n-1) + O(1)$ čije je rešenje takođe $O(n)$). I intuitivno je jasno da funkcija u svakom pozivu mora da obiđe i

levo i desno podstablo da bi odredila visinu, tako da funkcija obilazi sve čvorove (jednom) i stoga je linearne složenosti u odnosu na broj čvorova stabla.

Sa ovom funkcijom na raspolaganju, faktor ravnoteže možemo izračunati veoma jednostavno.

```
// izracunava faktore ravnoteze svih cvorova
void izracunajFaktoreRavnoteze(cvor* koren) {
  if (koren != nullptr) {
     izracunajFaktoreRavnoteze(koren->levo);
     izracunajFaktoreRavnoteze(koren->desno);
     koren -> faktorRavnoteze =
       abs(visina(koren->levo) - visina(koren->desno));
  }
}
```

Pod pretpostavkom da je drvo koliko-toliko balansirano, složenost ove funkcije zadovoljava jednačinu $T(n) = 2 \cdot T(n/2) + O(n)$ i njena je složenost, znamo na osnovu master teoreme, $O(n \log n)$. Ako je drvo izdegenerisano u listu (što se može desiti u najgorem slučaju), dobija se da vreme izvršavanja zadovoljava jednačinu $T(n) = T(n-1) + O(n)$ čije je rešenje $O(n^2)$.

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. Bazu čini slučaj praznog stabla čija je visina nula i koje ne sadrži čvorove čiji faktor ravnoteže treba izračunati. Ako pretpostavimo da umemo da izračunamo faktore ravnoteže svih čvorova poddrveta, kao i njihove visine, tada faktor ravnoteže korena možemo jednostavno izračunati kao apsolutnu vrednost razlike tih visina, a visinu celog drveta kao broj za jedan veći od maksimuma visine levog i desnog poddrveta. Implementaciju je jednostavno napraviti.

```
// izracunava faktore ravnoteze svih cvorova i vraca visinu stabla
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;
}
```

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 (jer je rešenje jednačine $T(n) = T(n-1)+O(1)$ takođe $O(n)$).

Prvi algoritam drvo od milion nasumično odabranih elemenata obrađuje za oko $0,672$ sekundi, a drugi za oko 0.0731 sekundi. Za drvo degenerisano u listu od

10,000 elemenata prva implementacija izračunava faktore ravnoteže za $0,003$ sekunde, a druga za $0,133$ sekunde (usled razlike između linearne i kvadratne složenosti).

Red je primetiti i da se zbog velike cene dinamičke alokacije čvorova drvo od milion elemenata formira za oko $0,750$ sekundi i briše za $0,128$ sekundi, tako da su oba vremena izračunavanja faktora ravnoteže znatno manja od vremena kreiranja i brisanja drveta (pod pretpostavkom kolike-tolike balansiranosti koja je opravdana u slučaju ubacivanja nasumično odabranih elemenata opravdano je očekivati je da će složenost kreiranja drveta biti $O(n \log n)$, uz veliki konstantni faktor, dok će složenost brisanja uvek biti linearna $O(n)$). U slučaju izdegenerisanog drveta od 10,000 elemenata, formiranje traje oko $0,166$ sekundi, ne toliko zbog alokacije, koliko zbog kvadratne složenosti koja se javlja.

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 i odrediti njegovu vremensku složenost.

Problem se jednostavno rešava korišćenjem induktivno-rekurzivne konstrukcije. 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 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 (visina je jednaka broju grana na najdaljem putu između korena i lista).

```
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});
}
```

Slično kao i u prethodnom zadatku, složenost funkcije visina je $O(n)$, dok je složenost izračunavanja dijametra $O(n \log n)$ u slučaju balansiranog stabla tj. $O(n^2)$ u opštem slučaju.

Ponovo možemo ojačati induktivnu hipotezu i visinu izračunavati paralelno sa dijametrom.

```
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});
}
```

U ovom slučaju složenost najgoreg slučaja je $O(n)$, čak i kada drvo nije balansirano.

Segment maksimalnog zbira

Vratimo se sada problemu određivanja segmenta maksimalnog zbira koji smo već ranije razmatrali i prikažimo još nekoliko rešenja zasnovanih na induktivno-rekurzivnoj konstrukciji.

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

Kadanov algoritam (dinamičko programiranje)

Prikažimo sada još nekoliko algoritama linearne složenosti za rešavanje ovog problema. Možda najpoznatiji algoritam je Kadanov algoritam, koji se ubraja u algoritme dinamičkog programiranja (o kojima će više biti reči kasnije).

Pokušavamo da algoritam zasnujemo na induktivnoj konstrukciji. Za prazan niz, jedini segment je prazan i njegov je zbir nula (to je ujedno najveći zbir koji se može dobiti). Smatramo da umemo da problem rešimo za proizvoljan niz dužine n i na osnovu toga pokušavamo da rešimo zadatak za niz dužine $n + 1$ (polazni niz proširen jednim dodatnim elementom). 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 (uključujući i mogućnost da je tu i prazan segment). 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 (krenuvši od praznog sufiksa, pa dodajući unazad jedan po jedan element) najviše što možemo dobiti je algoritam kvadratne složenosti (probajte da se uverite da je to zaista tako). 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 (tj. najvećeg sufiksa niza pre proširenja). Naime, ako je zbir najvećeg segmenta koji se završava na prethodnoj poziciji i tekućeg elementa pozitivan, onda je to upravo najveći zbir sufiksa proširenog niza (prazan segment ima zbir nula, pa je njegov zbir manji od pronađenog pozitivnog zbira segmenta, a neprazni sufiksi moraju da uključe tekući element i neki sufiks niza pre proširenja, pa nam je jasno da zbir tog sufiksa mora biti određen optimalno). 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.

Ako bismo formirali rekurzivnu funkciju koja vrši takvo izračunavanje, dobili bismo neefikasan algoritam jer bi se isti pozivi ponavljali više puta. Umesto toga možemo napraviti iterativan algoritam kome je invarijanta da u svakom koraku petlje znamo ove dve vrednosti (maksimum segmenta i maksimum sufiksa).

```
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;
```

Analiza složenosti je veoma jednostavna. Telo petlje koje je konstantne složenosti se izvršava tačno n puta, pa je složenost ovog algoritma takođe $O(n)$.

Zbirovi prefiksa (parcijalne sume niza)

Još jedan algoritam kojim možemo efikasno rešiti ovaj zadatak se zasniva na korišćenju *zbirova prefiksa*. Naime, ako znamo zbir svakog prefiksa niza, onda zbir svakog segmenta možemo dobiti kao razliku zbirova dva prefiksa. Naime, važi da je

$$\sum_{i=l}^{d} a_i = \sum_{i=0}^{d} a_i - \sum_{i=0}^{l-1} a_i$$

Računamo da je zbir praznog segmenta $\sum_{i=0}^{-1} a_i$ po definiciji jednak nuli.

Obratite pažnju na to da je ova tehnika zapravo analogon Njutn-Lajbnicove formule koju ste sretali tokom izučavanja matematičke analize.

Ponovo koristimo induktivnu konstrukciju iz prethodnog zadatka i niz proširujemo za jedan po jedan element. Da bismo umeli da izračunamo maksimum segmenta proširenog niza potrebno je da znamo maksimum segmenta polaznog niza i da izračunamo maksimalni sufiks proširenog niza. Na osnovu razlaganja na zbirove prefiksa, maksimalni zbir sufiksa proširenog niza se dobija kao razlika zbira celog proširenog niza (tj. zbira prefiksa do tekuće pozicije) i zbira nekog prefiksa neproširenog niza (prazan sufiks ne moramo analizirati, jer je prazan niz već uzet u obzir u sklopu baze indukcije). Pošto je umanjenik konstantan, da bismo maksimizovali razliku potrebno da znamo najmanji mogući umanjilac, tj. da znamo najmanji zbir prefiksa koji se završava na nekoj poziciji ispred tekuće. I tekući i minimalni zbir prefiksa možemo održavati inkrementalno. Kada niz proširimo jednim elementom, zbir prefiksa uvećavamo za taj element, poredimo ga sa dotadašnjim minimalnim zbirom prefiksa i ako je manji, ažuriramo minimalni zbir prefiksa. Naravno, održavamo i globalni maksimalni zbir segmenta koji ažuriramo svaki put kada naiđemo na segment (sufiks) čiji je zbir veći od dotadašnjeg maksimuma.

Dakle, i u ovom rešenju je induktivna hipoteza pojačana i pretpostavljamo da pored segmenta najvećeg zbira u obrađenom delu niza umemo da odredimo i minimalni zbir prefiksa obrađenog dela niza.

```
int zbirPrefiksa = 0;
int minZbirPrefiksa = zbirPrefiksa;
int maxZbir = 0;
for (int i = 0; i < n; i++) {
  zbirPrefiksa += a[i];
  int zbirSufiksa = zbirPrefiksa - minZbirPrefiksa;
  if (zbirSufiksa > maxZbir)
    maxZbir = zbirSufiksa;
  if (zbirPrefiksa < minZbirPrefiksa)
    minZbirPrefiksa = zbirPrefiksa;
}
cout << maxZbir << endl;
```

Analiza složenosti je veoma jednostavna. Telo petlje koje je konstantne složenosti se izvršava tačno n puta, pa je složenost ovog algoritma takođe $O(n)$.

Zbirovi prefiksa - maksimalni zbir segmenta deljiv sa k

Rešenje sa zbirovima prefiksa je pogodno, jer se može uopštiti i na neke srodne probleme. Razmatrajmo naredno uopštenje polaznog zadatka.

Problem: Dat je niz od n nenegativnih celih brojeva. Definisati efikasan algoritam koji određuje najveći zbir koji ima neki njegov segment (podniz uzastopnih elemenata) koji je deljiv datim brojem k.

Invarijanta petlje je da za svaku vrednost i za svaki ostatak o od 0 do $k-1$ u nizu `minZbirPrefiksaModK` pamtimo najmanji zbir svih prefiksa oblika $[0, j)$ za $0 \leq j \leq i$ koji pri deljenju sa k daje ostatak o tj. -1 ako takav prefiks ne postoji, da u promenljivoj `zbirPrefiksa` čuvamo vrednost prefiksa $[0, i)$, a da u promenljivoj `maxZbir` čuvamo maksimalni zbir svih segmenata $[a, b]$ za $0 \leq a \leq b < i$.

Za $i = 0$ jedini prefiks je $[0, 0)$ tj. prazan prefiks čiji je zbir 0 (on pri deljenju sa k daje ostatak 0). Za ostale ostatke ne postoji prefiks koji se završava do pozicije i čiji bi zbir dao te ostatke. Zato niz `minZbirPrefiksaModK` inicijalizujemo tako da na poziciju 0 upisujemo 0, a na ostale pozicije upisujemo -1. Promenljivu `zbirPrefiksa` inicijalizujemo na nulu, isto kao i `maxZbir` (prazan prefiks ima zbir nula, koji jeste deljiv sa k).

Nakon tela petlje vrši se uvećanje promenljive `i` za jedan tako da je $i' = i + 1$. Promenljiva `zbirPrefiksa` je na ulazu u petlju sadržala vrednost zbira svih elemenata na pozicijama $[0, i)$, i da bi nakon izvršavanja tela petlje sadržala vrednosti na pozicijama $[0, i') = [0, i]$ potrebno je uvećati je za a_i. Od svih prefiksa oblika $[0, j)$, za $j \leq i'$ jedini koji ranije nije razmatran je $[0, i')$, čiji je zbir sadržan u promenljivoj `zbirPrefiksa` nakon njenog uvećanja. Niz `minZbirPrefiksaModK` je potrebno ažurirati u skladu sa pojavom tog novog prefiksa. Jedino mesto u nizu koje se može promeniti je ono koje odgovara ostatku pri deljenju tog zbira sa k. Ako na tom mestu u nizu stoji -1 znači da je ovo prvi prefiks čiji zbir pri deljenju sa k daje baš taj ostatak, pa je ujedno i najmanji i u skladu sa tim na mesto tekućeg ostatatka upisujemo tekući zbir prefiksa. Ako ne stoji -1, znači da se ranije već javljao neki prefiks čiji zbir daje isti ostatak. Pošto su elementi polaznog niza po pretpostavci nenegativni, taj kraći prefiks mora da ima manji (ili eventualno jednak) zbir, pa u tom slučaju niz najmanjih zbirova prefiksa ne menjamo. U tom slučaju znamo da smo našli dva različita prefiksa koji daju isti ostatak, pa smo našli i segment (sufiks koji se završava na poziciji i) čiji je zbir deljiv brojem k, pa je potrebno taj sufiks još uporediti sa tekućim maksimumom i ako je potrebno ažurirati maksimum.

```
vector<int> minZbirPrefiksaModK(k, -1);
minZbirPrefiksaModK[0] = 0;

int zbirPrefiksa = 0;
int maxZbir = zbirPrefiksa;
for (int i = 0; i < n; i++) {
  zbirPrefiksa += a[i];

  if (minZbirPrefiksaModK[zbirPrefiksa % k] == -1)
      minZbirPrefiksaModK[zbirPrefiksa % k] = zbirPrefiksa;
  else {
    int zbirSufiksa =
        zbirPrefiksa - minZbirPrefiksaModK[zbirPrefiksa % k];
    if (zbirSufiksa > maxZbir)
       maksZbir = zbirSufiksa;
```

```
  }
}

cout << maxZbir << endl;
```

Za vežbu pokušajte da napišete program koji određuje koliko ima segmenata u nizu prirodnih brojeva čiji je zbir paran broj.

Broj rastućih segmenata

Vratimo se na problem određivanja broja rastućih segmenata datog niza prirodnih brojeva.

U prethodnim algoritmima u kojima smo analizirali sve segmente, prvo smo analizirali broj rastućih segmenata koji počinju na poziciji 0, zatim broj rastućih segmenata koji počinju na poziciji 1, zatim na poziciji 2, itd., sve do pozicije $n-2$ (na poziciji $n-1$ ne može počinjati ni jedan rastući segment, jer zahtevamo da su segmenti bar dvočlani).

Problem možemo rešiti i 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 2, zatim na poziciji 3, itd., sve do pozicije $n-1$ (na poziciji 0 se ne može završiti rastući segment, jer zahtevamo da su segmenti bar dvočlani).

Rešenje se zasniva na tome da broj rastućih 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 $[a_{j-1}, a_j]$.

Dakle, 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 upravo na poziciji j.

Na osnovu ovog zapažanja možemo izgraditi naredni, veoma efikasni algoritam.

```
// ukupan broj rastucih serija
int ukupanBrojRastucih = 0;
// broj rastucih koji se zavrsavaju na tekucoj poziciji
int tekuciBrojRastucih = 0;
for (int j = 1; j < n; i++) {
  if (a[j] > a[j-1]) {
    // tekuci element produzava sve rastuce segmente koji su
    // se zavrsili na prethodnoj poziciji i dodaje jos jedan
    // nov dvoclan rastuci segment
```

```
    tekuciBrojRastucih++;
    // dodajemo broj rastucih koji se zavrsavaju na poziciji i na
    // ukupan broj rastucih segmenata
    ukupanBrojRastucih += tekuciBrojRastucih;
  } else {
    // na tekucoj poziciji se ne zavrsava ni jedan rastuci segment
    tekuciBrojRastucih = 0;
  }
}
```

Složenost ovog postupka je očigledno $O(n)$.

Da zaključimo, videli smo nekoliko različitih algoritama za rešavanje nekih odabranih problema, koji su se razlikovali po svojoj složenosti. Ključna opaska je da su mnogi od njih nastali korišćenjem prilično standardnog dijapazona alatki koje dobar algoritmičar ima u svom arsenalu: tehnika grube sile, optimizacija izvođenjem eksplicitne formule, optimizacija sortiranjem, optimizacija na osnovu inkrementalnosti, tehnika dva pokazivača, induktivno-rekurzivna konstrukcija, tehnika dinamičkog programiranja, ojačavanje induktivne hipoteze, zbirovi prefiksa, tehnika podeli-pa-vladaj, ... U nastavku kursa ćemo se detaljnije fokusirati na svaku od njih i ilustrovaćemo ih sa dosta konkretnih primera. Naravno, uvešćemo još mnoge korisne tehnike koje nismo ilustrovali na ovom zadatku. Kada naučite te opšte tehnike imaćete veoma moćan arsenal oružija kojim ćete moći da napadate algoritamske probleme i da dolazite do veoma elegantnih i efikasnih rešenja. Dakle, konstrucija algoritama se da naučiti i prilično šablonizovati, ali je u okviru primene svakog šablona potrebno dosta razmišljanja!

Maksimalna suma nesusednih elemenata pozitivnog niza

Problem: Dat je niz nenegativnih celih brojeva a. Napisati program koji određuje najveći zbir podniza datog niza koji ne sadrži dva uzastopna člana niza a. Na primer, za niz $7, 3, 2, 4, 1, 5$ najveći takav podniz je $7, 4, 5$, čiji je zbir 16.

Pokušamo da zadatak rešimo induktivno-rekurzivnom konstrukcijom. 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 (jer je uvek nenegativan) doda se na maksimalni zbir elemenata prefiksa koji ne uključuje pretposlednji element i drugog koji se dobija kao zbir elemenata prefiksa koji uključuje pretposlednji element. Dakle, pretpostavljamo da za prefiks znamo maksimalni zbir bez njegovog poslednjeg i sa njegovim poslednjim elementom. Zato ojačavamo induktivnu hipotezu i pretpostavljamo da za svaki prefiks niza umemo da odredimo upravo te dve vrednosti. Bazu indukcije čini jednočlani prefiks niza. Maksimalni zbir sa uključenim njegovim jedinim elementom jednak je tom elementu, dok je maksimalni zbir bez njega jednak nuli. 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.

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

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

  // prvi element se ucitava van petlje zbog inicijalizacije
  // to je ujedno i baza indukcije, jednoclani prefiks niza
  int x;
  cin >> x;
  int maks_zbir_bez = 0;
  int maks_zbir_sa = x;

  // u petlji obradjujemo tekuci element
  for (int i = 1; i < n; i++) {
    int x;
    cin >> x;
    int maks_zbir_prethodne_pozicije =
      max(maks_zbir_sa, maks_zbir_bez);
    maks_zbir_sa = maks_zbir_bez + x;
    maks_zbir_bez = maks_zbir_prethodne_pozicije;
  }

  cout << max(maks_zbir_bez, maks_zbir_sa) << endl;

  return 0;
}
```

Složenost ovog algoritma je prilično očigledno $O(n)$.

Čas 4.1, 4.2, 4.3 - Bibliotečke strukture podataka

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* (ovaj pojam ne treba mešati sa strukturama u programskom jeziku C).

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 (obično asimptotsku ili amortizovanu) 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. Najznačajnije strukture podataka su implementirane u bibliotekama savremenih programskih jezika (C++, Java, C#, Python i mnogi drugi imaju veoma bogate biblioteke struktura podataka). U ovom poglavlju ćemo se baviti upotrebom i primenom gotovih struktura podataka u implementaciji nekih interesantnih algoritama. Sa druge strane, poznavanje implementacije samih struktura podataka može biti značajno za njihovo duboko razumevanje, ali i za mogućnost njihovog prilagođavanja upotrebi u nekim specifičnim algoritmima, kao i za mogućnost definisanja novih struktura podataka, koje nisu podržane bibliotekom programskog jezika koji se koristi. Stoga ćemo se u narednom poglavlju baviti načinima implementacije struktura podataka.

Skalarni tipovi

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).

Parovi, torke, slogovi

Najjednostavnije kolekcije podataka sadrže samo mali broj (2, 3, 4, ...) podataka, koji mogu biti i različitih tipova. U savremenim programskim jezicima postoje dva načina da se podaci upakuju na taj način. Jedno su uređeni parovi (engl. pair) tj. uređene torke (engl. tuple) gde se svakom pojedinačnom podatku pristupa na osnovu pozicije. Drugo su slogovi tj. strukture (engl. record, struct) u gde se svakom pojedinačnom podatku pristupa 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 koje obično teče ad hoc.

Parovi

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 (npr. `pair<int, int> p{0, 0}`). Funkcijom `tie` moguće je par razložiti na komponente (npr. `tie(x, y) = p` uzrokuje da promenljiva `x` sadrži prvu, a `y` drugu komponentu para). Prikažimo sada upotrebu parova u jeziku C++ za modelovanje tačaka u ravni.

Problem: Date su koordinate tri tačke trougla. Odredi koordinate njegovog središta.

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

int main() {
   pair<double, double> A, B, C;
   cin >> A.first >> A.second;
   cin >> B.first >> B.second;
   cin >> C.first >> C.second;
   pair<double, double> T;
   T.first = (A.first + B.first + C.first) / 3.0;
   T.second = (A.first + B.first + C.first) / 3.0;
   cout << T.first << " " << t.second << endl;
   return 0;
}
```

U jeziku Python, upotreba parova je još jednostavnija.

```
(Ax, Ay) = (int(input()), int(input()))
(Bx, By) = (int(input()), int(input()))
(Cx, Cy) = (int(input()), int(input()))
(Tx, Ty) = ((Ax + Bx + Cx) / 3.0, (Ay + By + Cy) / 3.0)
print(Tx, Ty)
```

Torke

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 (i tada se koristi leksikografski poredak). Prikažimo sada upotrebu torki u jeziku C++ za modelovanje poređenja datuma (koje je suštinski leksikografsko).

Problem: Sa ulaza se učitava niz datuma. Odredi najkasniji od njih.

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

```
tuple<int, int, int> ucitaj_datum() {
   int d, m, g;
   cin >> d >> m >> g;
   return make_tuple(g, m, d);
}

void ispisi_datum(const tuple<int, int, int>& datum) {
   cout << get<2>(datum) << " " <<
           get<1>(datum) << " " <<
           get<0>(datum) << endl;
}

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

   tuple<int, int, int> max_datum = ucitaj_datum();
   for (int i = 1; i < n; i++) {
      tuple<int, int, int> datum = ucitaj_datum();
      if (datum > max_datum)
         max_datum = datum;
   }
   ispisi_datum(max_datum);
   return 0;
}
```

Korišćenje torki u jeziku Python je još jednostavnije.

```
def ucitaj_datum():
    (d, m, g) = (int(input()), int(input()), int(input()))
    return (g, m, d)

def ispisi_datum(datum):
    (g, m, d) = datum
    print(d, m, g)

n = int(input())
max_datum = ucitaj_datum()
for i in range(1, n):
    datum = ucitaj_datum()
    if datum > max_datum:
       max_datum = datum
ispisi_datum(max_datum)
```

Slogovi (strukture)

Nema velike razlike između definisanja i korišćenja struktura u jeziku C i jeziku C++. Ilustrujmo to na primeru razlomaka.

Problem: Definisati strukturu za predstavljanje razlomaka, funkciju za sabiranje razlomaka i glavni program koji ih testira.

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

struct razlomak {
   int brojilac, imenilac;
};

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

void skrati(razlomak& r) {
   int n = nzd(r.brojilac, r.imenilac);
   r.brojilac /= n;
   r.imenilac /= n;
}

razlomak saberi(const razlomak& r1, const razlomak& r2) {
   razlomak zbir;
   zbir.brojilac = r1.brojilac * r2.imenilac + r2.brojilac * r1.imenilac;
   zbir.imenilac = r1.imenilac * r2.imenilac;
   skrati(zbir);
   return zbir;
}

int main() {
   razlomak r1{3, 5}, r2{2, 3};
   razlomak zbir = saberi(r1, r2);
   cout << zbir.brojilac << "/" << zbir.imenilac << endl;
   return 0;
}
```

Nizovi (statički, dinamički)

Nizovi (u raznim oblicima) predstavljaju praktično osnovne kolekcije podataka. Osnovna karakteristika nizova je to da omogućavaju efikasan pristup (po pravilu u složenosti $O(1)$ ili bar u amortizovanoj složenosti $O(1)$) elementu niza na osnovu njegove pozicije (kažemo i indeksa). Pristup van granica niza obično prouzrokuje grešku u programu (u nekim jezicima poput Java, C# ili Python se podiže izuzetak, a u nekim poput C ili C++ ponašanje nije definisano). U zavis-

nosti 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. U velikom broju savremenih programskih jezika (npr. C++, Java, C#, Python) brojanje elemenata počinje od nule.

Statički nizovi

Rad sa statičkim nizovima u jeziku C++ je veoma sličan radu sa statičkim nizovima u C-u. Ilustrujmo to jednim jednostavnim primerom.

Problem: Učitava se najviše 100 brojeva. Izračunaj element koji najmanje odstupa od proseka.

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

int main() {
   // učitavamo broj elemenata i zatim sve elemente u niz
   int a[100];
   int n;
   cin >> n;
   for (int i = 0; i < n; i++)
      cin >> a[i];

   // izračunavamo prosek
   int zbir = 0;
   for (int i = 0; i < n; i++)
      zbir += a[i];
   double prosek = (double) zbir / (double) n;

   // određujemo i ispisujemo element koji najmanje
   // odstupa pod proseka
   int min = 0;
   for (int i = 1; i < n; i++)
      if (abs(a[i] - prosek) < abs(a[min] - prosek))
         min = i;
   cout << a[i] << endl;

   return 0;
}
```

Naglasimo da je u ovom zadatku bilo neophodno elemente prilikom učitavanja memorisati, jer su nam za rešenje zadatka potrebna dva prolaska kroz podatke (prvi u kom se određuje prosek i drugi u kom se određuje minimalno odstupanje od proseka). Da se tražilo samo određivanje proseka ili prosečno odstupanje od

minimuma, niz nije bilo neophodno koristiti (razmisli kako bi se to uradilo u memorijskoj složenosti $O(1)$).

Dinamički nizovi

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 (a taj broj kasnije može biti menjan). 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 (tj. broja njegovih elemenata). 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 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).

Prikažimo upotrebu vektora na sledećoj varijaciji prethodnog zadatka.

Problem: Učitavaju se brojevi do kraja standardnog ulaza. Koji element najmanje odstupa od proseka?

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

int main() {
   // učitavamo sve elemente u vektor
   vector<int> a;
   string linija;
   while (cin >> linija)
      a.push_back(stoi(linija));

   // izračunavamo prosek
   int n = a.size();
   int zbir = 0;
   for (int i = 0; i < n; i++)
      zbir += a[i];
   double prosek = (double) zbir / (double) n;

   // određujemo i ispisujemo element koji najmanje
```

```
   // odstupa od proseka
   int min = 0;
   for (int i = 1; i < n; i++)
      if (abs(a[i] - prosek) < abs(a[min] - prosek))
         min = i;
   cout << a[min] << endl;

   return 0;
}
```

Iako se osnovne kolekcije u jeziku Python nazivaju liste, one zapravo predstavljaju dinamičke nizove (pre svega zbog efikasnog indeksnog pristupa i zbog mogućnosti dodavanja elemenata metodom `append`).

```
import sys

# učitavamo elemente u listu
a = []
for linija in sys.stdin:
    a.append(int(linija))

prosek = sum(a) / len(a)

min = 0;
for i in range(1, n):
    if abs(a[i] - prosek) < abs(a[min] - prosek):
        min = i

print(a[min])
```

U jeziku Java dinamički nizovi su podržani putem klase `ArrayList`, a u jeziku C# putem klase `List` (pri čemu u ovim jezicima ne postoje klasični statički nizovi, već se obični nizovi takođe alociraju na hipu, tokom izvršavanja programa, jedino što im nije moguće dinamički menjati dimenziju).

Višedimenzionalni nizovi, matrice (statički, dinamički)

U nekim slučajevima su nam potrebni višedimenzionalni nizovi. U nekim jezicima oni su podržani direktno, a nekada se realizuju kao nizovi nizova. 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). Ilustrujmo to jednim jednostavnim primerom.

Problem: Napisati program koji učitava i pamti matricu dimenzije 5×5 i zatim izračunava njen trag (zbir dijagonalnih elemenata).

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

int main() {
  // učitavamo matricu
  int A[5][5];
  for (int v = 0; v < 5; v++)
    for (int k = 0; k < 5; k++)
      cin >> A[v][k];

  // izračunavamo i ispisujemo trag
  int trag = 0;
  for (int i = 0; i < 5; i++)
    trag += A[i][i];
  cout << trag << endl;

  return 0;
}
```

Ukoliko dimenzija nije unapred poznata, najlakše nam je da za smeštanje matrice upotrebimo vektor vektora.

Problem: Napisati program koji učitava i pamti matricu dimenzije $n \times n$ i zatim izračunava njen trag.

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

int main() {
  // učitavamo elemente matrice
  int n;
  cin >> n;
  // alociramo prostor za n vrsta matrice
  vector<vector<int>> A(n);
  for (int v = 0; v < n; v++) {
     // u vrsti v alociramo prostor za n elemenata
     A[v].resize(n);
     // učitavamo elemente vrste v
     for (int k = 0; k < n; k++)
       cin >> A[v][k];
   }

   // izračunavamo i ispisujemo trag
   int trag = 0;
   for (int i = 0; i < n; i++)
     trag += A[i][i];
```

```
    cout << trag << endl;

    return 0;
}
```

U slučaju vektora vektora, vrste mogu imati i različit broj kolona.

Problem: Napisati program koji učitava i pamti ocene nekoliko studenata i za svakog izračunava prosečnu ocenu. Sa ulaza se prvo učitava broj studenata, a zatim za svakog studenta broj ocena i nakon toga pojedinačne ocene (broj ocena različitih studenata može biti različit).

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

int main() {
  // učitavamo tabelu ocena svih učenika
  int brojUcenika;
  cin >> brojUcenika;
  vector<vector<int>> ocene(brojUcenika);
  for (int u = 0; u < brojUcenika; u++) {
    int brojOcena;
    cin >> brojOcena;
    ocene[u].resize(brojOcena);
    for (int o = 0; o < n; o++)
      cin >> ocene[u][o];
  }

  // izračunavamo i ispisujemo sve proseke ocena
  for (int u = 0; u < brojUcenika; u++) {
    int zbir = 0;
    int brojOcena = ocene[u].size();
    for (int o = 0; o < brojOcena; o++)
      zbir += ocene[u][o];
    cout << (double) zbir / (double) brojOcena << endl;
  }

  return 0;
}
```

Stekovi

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. Podržane su sledeće 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).

U jeziku Python stek se simulira pomoću običnih lista. Metoda `append` dodaje element na kraj liste (koji se tumači kao vrh steka), dok metoda `pop` uklanja element sa steka.

Prikažimo upotrebu steka kroz nekoliko interesantnih primera.

Istorija veb-pregledača

Problem: Pregledač veba pamti istoriju posećenih sajtova i korisnik ima mogućnost da se vraća unatrag na sajtove koje je ranije posetio. Napisati program koji simulira istoriju pregledača tako što se učitavaju adrese posećenih sajtova (svaka u posebnom redu), a kada se učita red u kome piše `back` pregledač se vraća na poslednju posećenu stranicu.

Rešenje je krajnje jednostavno. Spisak posećenih sajtova čuvamo na steku. Naredbom `back` uklanjamo i ispisujemo sajt sa vrha steka (uz proveru da stek nije prazan).

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

int main() {
   stack<string> istorija;
   string linija;
   while (getline(cin, linija)) {
      if (linija == "back")
         if (!istorija.empty()) {
            cout << istorija.top() << endl;
            istorija.pop();
         } else {
            cout << "-" << endl;
         }
      else
         istorija.push(linija);
   }
```

```
   return 0;
}
```

U jeziku Python stek možemo simulirati pomoću najobičnije liste. Metodom `append` dodajemo element na kraj liste, a metodom `pop` skidamo element sa kraja.

```
import sys

istorija = []
for linija in sys.stdin:
    linija = linija.strip()
    if linija == "back":
        if istorija:
            print(istorija.pop())
        else:
            print("-")
    else:
        istorija.append(linija)
```

Problem: Napisati program koji sve učitane linije sa standardnog ulaza ispisuje u obratnom redosledu.

Jedno od mogućih rešenja je da se sve učitane linije smeste na stek, a da se zatim ispišu uzimajući jednu po jednu sa steka. Pošto stek funkcioniše po LIFO principu, redosled će biti obrnut (najkasnije dodata linija biće prva skinuta i ispisana, dok će prva postavljena linija biti skinuta i ispisana poslednja).

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

int main() {
   stack<string> s;
   string linija;
   while (cin >> linija)
      s.push(linija);
   while (!s.empty()) {
      cout << s.top() << endl;
      s.pop();
   }
   return 0;
}
```

Uparenost zagrada

Problem: Napisati program koji proverava da li su zagrade u infiksno zapisanom izrazu korektno uparene. Moguće je pojavljivanje malih, srednjih i velikih zagrada (tzv. običnih, uglastih i vitičastih).

Kada bismo radili samo sa jednom vrstom zagrada, dovoljno bi bilo samo održavati broj trenutno otvorenih zagrada. Pošto imamo više vrsta zagrada održavamo stek na kome pamtimo sve do sada otvorene, a nezatvorene zagrade. Kada naiđemo na otvorenu zagradu, stavljamo je na stek. Kada naiđemo na zatvorenu zagradu, proveravamo da li se na vrhu steka nalazi njoj odgovarajuća otvorena zagrada i skidamo je sa steka (ako je stek prazan ili ako se na vrhu nalazi neodgovarajuća otvorena zagrada, konstatujemo grešku). Na kraju proveravamo da li se stek ispraznio.

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

bool uparena(char oz, char zz) {
   return oz == '(' && zz == ')' ||
          oz == '[' && zz == ']' ||
          oz == '{' && zz == '}';
}

bool otvorena(char c) {
   return c == '(' || c == '[' || c == '{';
}

bool zatvorena(char c) {
   return c == ')' || c == ']' || c == '}';
}

int main() {
   string izraz;
   cin >> izraz;
   stack<char> zagrade;
   bool OK = true;
   for (char c : izraz) {
      if (otvorena(c))
        zagrade.push(c);
      else if (zatvorena(c)) {
         if (zagrade.empty() || !uparena(zagrade.top(), c)) {
            OK = false;
            break;
         }
         zagrade.pop();
      }
   }
   if (!zagrade.empty())
      OK = false;

   cout << (OK ? "tacno" : "netacno") << endl;
   return 0;
}
```

Vrednost 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.

Postfiksna notacija se ponekad naziva i poljska notacija, a postfiksna notacija se ponekad naziva i obratna poljska notacija (engl. reverse polish notation, RPN) u čast logičara Jana Lukašijeviča koji ju je izumeo. 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. Npr. za izraz `34+5*` na stek postavljamo 3, zatim 4, nakon toga te dve vrednosti uklanjamo i postavljamo 7, zatim postavljamo 5 i na kraju uklanjamo 7 i 5 i menjamo ih sa 35.

```
#include <iostream>
#include <string>
#include <stack>
#include <cctype>
using namespace std;

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;
}
```

Prevođenje potpuno zagradjenog infiksnog izraza u postfiksni oblik

Problem: Dat je ispravan infiksni aritmetički izraz koji ima zagrade oko svake primene binarnog operatora. Napisati program koji ga prevodi u postfiksni oblik. Na primer, `(((3*5)+(7+(2*1)))*4)` se prevodi u `35*721*++4*`. Jednostavnosti radi pretpostaviti da su svi operandi jednocifreni brojevi i da se javljaju samo operacije sabiranja i množenja.

Činjenica da je izraz potpuno zagrađen olakšava izračunavanje, jer nema potrebe da vodimo računa o prioritetu i asocijativnosti operatora.

Jedan način da se pristupi rešavanju problema je da se primeni induktivno-rekurzivni pristup. Obrada strukturiranog ulaza rekurzivnim funkcijama se naziva *rekurzivni spust* i detaljno se izučava u kursevima prevođenja programskih jezika. Definišemo rekurzivnu funkciju čiji je zadatak da prevede deo niske koji predstavlja ispravan infiksni izraz. On može biti ili broj, kada je prevođenje trivijalno jer se on samo prepiše na izlaz ili izraz u zagradama. U ovom drugom slučaju čitamo otvorenu zagradu, zatim rekurzivnim pozivom prevodimo prvi operand, nakon toga čitamo operator, zatim rekurzivnim pozivom prevodimo drugi operand, nakon toga čitamo zatvorenu zagradu i ispisujemo operator koji smo pročitali (on biva ispisan neposredno nakon prevoda svojih operanada).

Promenljiva `i` menja svoju vrednost kroz rekurzivne pozive. Primetimo da se ona prenosi po referenci tako da predstavlja i ulaznu i izlaznu veličinu funkcije. Zadatak funkcije je da pročita izraz koji počinje na poziciji i, da ga prevede u postfiksni oblik i da promenljivu `i` promeni tako da njena nova vrednost i' ukazuje na poziciju niske neposredno nakon izraza koji je preveden.

```
void prevedi(const string& izraz, int& i) {
  if (isdigit(izraz[i]))
     cout << izraz[i++];
  else {
    // preskačemo otvorenu zagradu
    i++;
    // prevodimo prvi operand
    prevedi(izraz, i);
    // pamtimo operator
    char op = izraz[i++];
    // prevodimo drugi operand
    prevedi(izraz, i);
    // preskačemo zatvorenu zagradu
    i++;
    // ispisujemo upamćeni operator
    cout << op;
```

```
  }
}

void prevedi(const string& izraz) {
  int i = 0;
  prevedi(izraz, i);
  cout << endl;
}
```

Da bismo se oslobodili rekurzije, potrebno je da upotrebimo stek. Ključna opaska je da se u stek okviru funkcije, pre rekurzivnog poziva za prevođenje drugog operanda pamti operator. Ovo nam sugeriše da nam je za nerekurzivnu implementaciju neophodno da održavamo stek na koji ćemo smeštati operatore. Kada naiđemo na broj prepisujemo ga na izlaz, kada naiđemo na operator stavljamo ga na stek, a kada naiđemo na zatvorenu zagradu skidamo i ispisujemo operator sa vrha steka.

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

Za vežbu vam ostavljamo da modifikujete ove funkcije tako da rezultat vrate u obliku niske umesto da ga ispisuju na standardni izlaz.

Vrednost potpuno zagradjenog infiksnog izraza

Problem: Dat je ispravan infiksni aritmetički izraz koji ima zagrade oko svake primene binarnog operatora. Na primer, `(((3*5)+(7+(2*1)))*4)` Napisati program koji izračunava njegovu vrednost (za izraz u zagradama to je vrednost 96). Jednostavnosti radi pretpostaviti da su svi operandi jednocifreni brojevi i da se javljaju samo operacije množenja i sabiranja.

Jedno rešenje je rekurzivno i u njemu bi se funkcija koja prevodi deo niske koji predstavlja ispravan izraz modifikovala tako da umesto prevođenja vraća njegovu vrednost.

```
int vrednost(string& izraz, int& i) {
  if (isdigit(izraz[i]))
```

```
    return izraz[i++] - '0';
  else {
    // preskačemo otvorenu zagradu
    i++;
    // čitamo i izračunavamo vrednost prvog operanda
    int op1 = vrednost(izraz, i);
    // pamtimo operator
    char op = izraz[i++];
    // čitamo i izračunavamo vrednost drugog operanda
    int op2 = vrednost(izraz, i);
    // preskačemo zatvorenu zagradu
    i++;
    // izračunavamo i vraćamo vrednost izraza
    return primeniOperator(op, op1, op2);
  }
}

int vrednost(string& izraz, int& i) {
  int i = 0;
  return vrednost(izraz, i);
}
```

Do rešenja možemo doći tako što iskombinujemo algoritam za prevođenje izraza u postfiksni oblik (pomoću steka na kom pamtimo operatore) i algoritma koji izračunava vrednost postfiksnog izraza (pomoću steka na kom pamtimo vrednosti). Kada bi se tokom prevođenja u postfiksni oblik na izlazu pojavio broj, taj broj ćemo postavljati na stek vrednosti. Kada bi se pojavio operator, sa vrha steka vrednosti ćemo skidati dva elementa, primenjivaćemo na njih odgovarajuću operaciju i rezultat vraćati na stek vrednosti.

Svaki podizraz je ili broj ili izraz u zagradama. Program ćemo organizovati tako da koristi dva steka (stek vrednosti i stek operatora). Nametnućemo uslov (invarijantu) da se neposredno nakon čitanja svakog broja i nakon svakog izraza u zagradama njihova vrednost nađe na vrhu steka vrednosti, pri čemu je stanje steka vrednosti ispod te vrednosti identično kao pre čitanja tog broja tj. izraza u zagradama, a da je stanje na steku operatora identično kao pre početka čitanja tog broja tj. izraza. Takođe, neposredno nakon čitanja operatora on treba da se nađe na vrhu steka operatora, pri čemu se ne menja stek vrednosti niti prethodni sadržaj steka operatora.

Krećemo od praznih stekova. Da bi se ova invarijanta održala, kada naiđemo na broj postavljaćemo ga na stek vrednosti. Slično, čim pročitamo operator, postavljaćemo ga na stek operatora. Najinteresantniji slučaj je kada naiđemo na zatvorenu zagradu. Neposredno pre te zagrade pročitan je drugi operand koji je ili broj ili izraz u zagradama i znamo da se njegova vrednost nalazi na vrhu steka vrednosti. Na osnovu invarijante znamo da se operator koji predstavlja operaciju u zagradi koja je upravo zatvorena nalazi na vrhu steka operatora (jer je nakon čitanja drugog operanda stek operatora u identičnom stanju kao pre njegovog čitanja, a to je stanje u kojem je upravo pročitan operator, pa je

postavljen na stek operatora). Neposredno pre tog operatora pročitan je prvi operand (koji je opet ili broj izraz u zagradama), tako da na osnovu invarijante znamo da se njegova vrednost nalazi kao druga sa vrha steka (ispod vrednosti drugog operanda). Zato skidamo dve vrednosti sa vrha steka vrednosti, skidamo operator sa vrha steka operatora, na osnovu tih elemenata izračunavamo novu vrednost i postavljamo je na stek vrednosti. Ovim smo postigli da se vrednost izraza u zagradama nalazi na vrhu steka vrednosti i da je stanje na stekovima ispod te vrednosti identično kao pre čitanja te otvorene zagrade. Pošto program čita jedan (isprvan) izraz u zagradama, kada se on pročita, znamo da se njegova vrednost nalazi na vrhu steka vrednosti.

```
int vrednost(const string& izraz) {
  stack<char> operatori;
  stack<int> vrednosti;
  for (char c : izraz) {
    if (isdigit(c)) {
      // stavljamo broj na vrh steka vrednosti
      vrednosti.push(c - '0');
    } else  if (c == ')') {
      // operator se nalazi na vrhu steka operatora
      char op = operatori.top(); operatori.pop();
      // operandi se nalaze na vrhu steka vrednosti
      int op2 = vrednosti.top(); vrednosti.pop();
      int op1 = vrednosti.top(); vrednosti.pop();
      // izračunavamo vrednost izraza u zagradi
      int v = primeniOperator(op, op1, op2);
      // postavljamo ga na stek vrednosti
      vrednosti.push(v);
    } else if (jeOperator(c)){
      // stavljamo operator na vrh steka operatora
      operatori.push(c);
    }
  }
  return vrednosti.top();
}
```

Prikažimo rad ovog algoritma na primeru izraza `(((3*5)+(7+(2*1)))*4)`.

```
stek operatora        stek vrednosti         ulaz
                                             (((3*5)+(7+(2*1)))*4)
                                              ((3*5)+(7+(2*1)))*4)
                                               (3*5)+(7+(2*1)))*4)
                                                3*5)+(7+(2*1)))*4)
                            3                    *5)+(7+(2*1)))*4)
     *                      3                     5)+(7+(2*1)))*4)
     *                      3 5                    )+(7+(2*1)))*4)
                            15                      +(7+(2*1)))*4)
     +                      15                       (7+(2*1)))*4)
     +                      15                        7+(2*1)))*4)
```

```
+                15 7                +(2*1)))*4)
+ +              15 7                 (2*1)))*4)
+ +              15 7                  2*1)))*4)
+ +              15 7 2                *1)))*4)
+ + *            15 7 2                 1)))*4)
+ + *            15 7 2 1               )))*4)
+ +              15 7 2                  ))*4)
+                15 9                    )*4)
                 24                      *4)
*                24                       4)
*                24 4                     )
                 96
```

Prevođenje infiksnog u postfiksni izraz

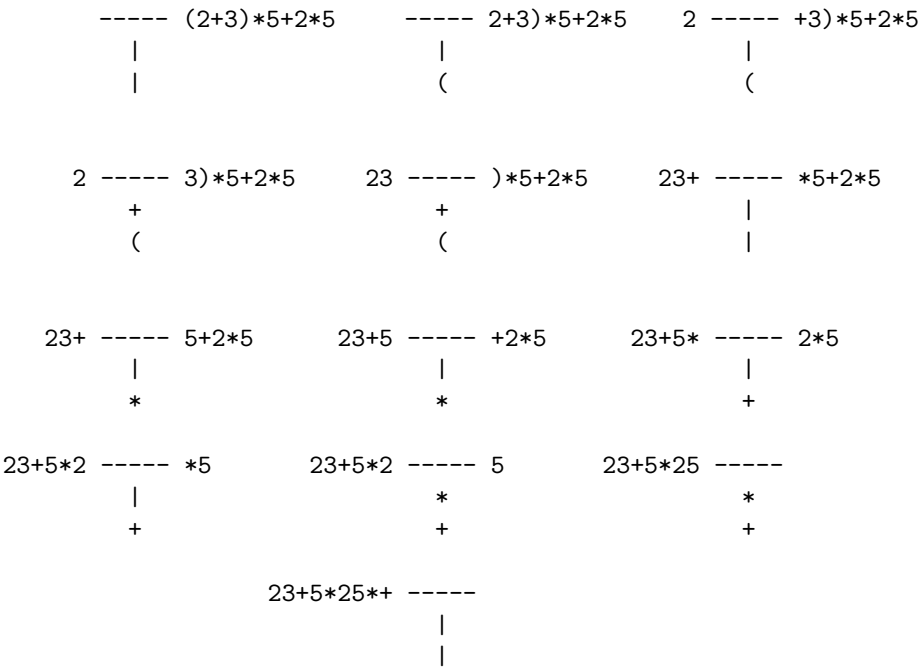
Problem: Napisati program koji vrši prevođenje ispravnog infiksno zadatog izraza u njemu ekvivalentan postfiksno zadati izraz. Pretpostaviti da su svi brojevi jednocifreni. Na primer, za izraz `2*3+4*(5+6)` program treba da ispiše `23*456+*+`.

Problem se rešava slično kao kod potpuno zagrađenih izraza, ali ovaj put se mora obraćati pažnja na prioritet i asocijativnost operatora. Rešenje se može napraviti rekurzivnim spustom, ali se time nećemo baviti u ovom kursu. 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 (bilo broja bilo izraza u zagradama), 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 (ako `op2` ima veći prioritet ili ako je prioritet isti, a asocijativnost desna) 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 na engleskom jeziku *Shunting yard algorithm*, što bi se moglo slobodno prevesti kao algoritam sortiranja železničkih vagona. Zamislimo da izraz treba da pređe sa jednog na drugi kraj pruge. Na pruzi se nalazi sporedni kolosek (pruga je u obliku slova T i sporedni kolosek je uspravna crta). Delovi izraza prelaze sa desnog na levi kraj (zamislimo da idu po gornjoj ivici slova T). Brojevi uvek prelaze direktno. Operatori se uvek zadržavaju na sporednom koloseku, ali tako da se pre nego što operator uđe na sporedni kolosek sa njega na izlaz prebacuju svi operatori koji su višeg prioriteta u odnosu na tekući ili imaju isti prioritet kao tekući a levo su asocijativni. I otvorene zagrade se postavljaju na sporedni kolosek, a kada naiđe zatvorena zagrada sa sporednog koloseka se uklanjaju svi operatori do otvorene zagrade. Kada se iscrpi ceo izraz na desnoj strani,

svi operatori sa sporednog koloseka se prebacuju na levu stranu. Jasno je da sporedni kolosek ima ponašanje steka, tako da implementaciju možemo napraviti korišćenjem steka na koji ćemo stavljati operatore.

Ilustrujmo ovu železničku analogiju jednim primerom.



Kada se algoritam razume, implementaciju je prilično jednostavno napraviti.

```
// prioritet operatora
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 == ')') {
       // prebacujemo na izlaz sve operatore unutar zagrade
       while (operatori.top() != '(') {
```

```
        cout << operatori.top();
        operatori.pop();
      }
      // uklanjamo otvorenu zagradu
      operatori.pop();
    }
    if (jeOperator(c)) {
      // prebacujemo na izlaz sve prethodne operatore višeg prioriteta
      while (!operatori.empty() && jeOperator(operatori.top()) &&
             prioritet(operatori.top()) >= prioritet(c)) {
        cout << operatori.top();
        operatori.pop();
      }
      // stavljamo operator na stek
      operatori.push(c);
    }
  }
  // prebacujemo na izlaz sve preostale operatore
  while (!operatori.empty()) {
    cout << operatori.top();
    operatori.pop();
  }
  cout << endl;
  return 0;
}
```

Vrednost 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`.

Na kraju, izračunavanje vrednosti izraza je opet kombinacija prevođenja u postfiksni oblik i izračunavanja vrednosti postfiksnog izraza.

```
// primenjuje datu operaciju na dve vrednosti na vrhu steka,
// zamenjujući ih sa rezultatom primene te operacije
void primeni(stack<char>& operatori, stack<int>& vrednosti) {
  // operator se nalazi na vrhu steka operatora
  char op = operatori.top(); operatori.pop();
  // operandi se nalaze na vrhu steka operatora
  int op2 = vrednosti.top(); vrednosti.pop();
  int op1 = vrednosti.top(); vrednosti.pop();
  // izracunavamo vrednost izraza
  int v;
  if (op == '+') v = op1 + op2;
  if (op == '*') v = op1 * op2;
  // postavljamo ga na stek operatora
```

```
  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 == ')') {
      // izračunavamo vrednost izraza u zagradi
      while (operatori.top() != '(')
        primeni(operatori, vrednosti);
    } else if (jeOperator(c)) {
       // obrađujemo sve prethodne operatore višeg prioriteta
       while (!operatori.empty() && jeOperator(operatori.top()) &&
              prioritet(operatori.top()) >= prioritet(c))
          primeni(operatori, vrednosti);
       operatori.push(c);
    }
  }

  // izračunavamo sve preostale operacije
  while (!operatori.empty())
    primeni(operatori, vrednosti);

  return vrednosti.top();
}
```

Nerekurzivno brzo sortiranje

Problem: Implementiraj brzo sortiranje nerekurzivno.

Jedna veoma važna upotreba steka je za realizaciju rekurzije. Kao što znamo, tokom izvršavanja rekurzivnih funkcija, na stek se smeštaju vrednosti lokalnih promenljivih i argumenata svakog aktivnog poziva funkcije. Rekurziju uvek možemo ukloniti i umesto sistemskog steka možemo ručno održavati stek sa tim podacima.

Prikažimo ovu tehniku uklanjanja rekurzije na primeru nerekurzivne implementacije algoritma brzog sortiranja. Recimo i da je ova tehnika opšta i da se rekurzija uvek može eliminisati na ovaj način. Za razliku od toga, eliminisanje specifičnih oblika rekurzije (poput, na primer, repne) nije uvek primenljivo, ali kada jeste, dovodi do bolje memorijske (pa i vremenske) efikasnosti jer se ne

koristi stek.

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.

```
#include <iostream>
#include <vector>
#include <stack>
#include <utility>
using namespace std;

int main() {
  // niz koji se sortira
  vector<int> a{3, 5, 4, 2, 6, 1, 9, 8, 7};
  // stek na kome čuvamo argumente rekurzivnih poziva
  stack<pair<int, int>> sortirati;
  // sortiranje kreće od obrade celog niza tj. pozicija (0, n-1)
  sortirati.push(make_pair(0, a.size() - 1));
  while (!sortirati.empty()) {
    // skidamo par (l, d) sa vrha steka
    auto p = sortirati.top();
    int l = p.first, d = p.second;
    sortirati.pop();
    // obrađujemo par (l, d) na isti način kao u rekurzivnoj implementaciji
    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]);
    // umesto rekurzivnih poziva njihove argumente
    // postavljamo na stek
    sortirati.push(make_pair(l, k-1));
    sortirati.push(make_pair(k+1, d));
  }
  // ispisujemo sortirani niz
  for (int x : a)
    cout << x << endl;
  return 0;
}
```

DFS nerekurzivno

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

Oslobađanje od rekurzije i ovaj put teče na isti način kao u prethodnom slučaju (na stek argument glavnog poziva i dok se stek ne isprazni na umesto rekurzivnih poziva na stek stavljamo njihove argumente).

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

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);
    }
  }
}


int main() {
  dfs(0);
  return 0;
}
```

U nastavku ćemo razmotriti još jednu familiju algoritama koji se implementiraju uz pomoć steka.

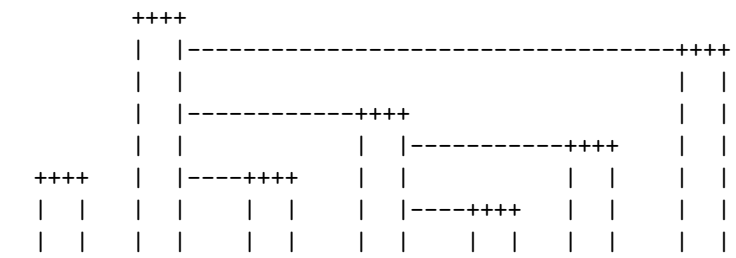
Najbliži veći prethodnik

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 nabliža poziciji i (tj. od svih pozicija levo od i

na kojima se nalaze elementi koji su strogo veći od a_i, j je najveća). Za svaku pronađenu poziciju j ispisati element a_j, a ako takva pozicija j ne postoji (ako su levo od i svi elementi manji ili jednaki a_i, tada ispisati karakter -). Takav element zvaćemo *najbliži veći prethodnik*. Na primer, za niz $3, 7, 4, 2, 6, 5$, ispisati $-, -, 7, 4, 7, 6$. Voditi računa o prostornoj i vremenskoj efikasnosti.

Naglasimo da nema nikakve značajne razlike da li se traži najbliži veći prethodnik, najbliži manji prethodnik, najbliži prethodnik koji je manji ili jednak ili najbliži prethodnik koji je veći ili jednak. Isti algoritam se može primeniti na bilo koju relaciju poretka.

Jedna lepa interpretacija prethodnog zadatka je sledeća. Neka niz a sadrži visine zgrada poređanih duž jedne ulice. Betmen želi da postavi horizontalno uže na svaku zgradu koje će ga dovesti do neke prethodne zgrade. Za svaku zgradu je potrebno ispisati visinu njoj prethodne zgrade do koje će voditi horizontalno uže sa tekuće zgrade. Ako su visine zgrada redom $2, 7, 2, 4, 1, 3, 6$, tada užad vode do zgrada čije su visine $-, 6, 6, 4, 4, 6$.



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

```
for (int i = 0; i < n; i++) {
  bool nadjen = false;
  for (int j = i-1; j >= 0; j--)
    if (a[j] > a[i]) {
       cout << a[j] << endl;
       nadjen = true;
       break;
    }
  if (!nadjen)
    cout << "-" << endl;
}
```

Pokušajmo da efikasniji algoritam konstruišemo induktivno-rekurzivnom konstrukcijom.

- Bazu čini jednočlan niz i sigurni smo da početni element nema prethodnika većeg od sebe (jer uopšte nema prethodnika). Sa prve zgrade nije moguće razvući kanap.

- 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 (elementa na poziciji k). Analizu krećemo od direktnog prethodnika tekućeg elementa (za poziciju k, analizu krećemo od pozicije $k-1$). Ako je taj element strogo veći od tekućeg, on mu je najbliži veći prethodnik (i do njega razvlačimo kanap). 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.

Ovaj algoritam možemo rekurzivno izraziti na sledeći način.

```
// funkcija vraća poziciju najbližeg većeg prethodnika elementa
// a[k], tj. -1 ako a[k] nema većih prethodnika
int najblizi_veci_prethodnik(int a[], int k) {
  // početni element nema prethodnika
  if (k == 0)
    return -1;
  // pretragu počinjemo od neposrednog prethodnika tekućeg elementa
  int p = k-1;
  // dok god je prethodnik unutar niza, ali nije strogo veći od tekućeg
  while (p != -1 && a[p] <= a[k])
    // prelazimo na analizu njegovog prethodnika
    p = najblizi_veci_prethodnik(a, p);
  // petlja se završila ili kada je p=-1 pa element nema prethodnika
  // ili kada je a[p] > a[k], pa je a[p] najbliži veći prethodnik elementu a[k]
  return p;
}

int main() {
  // učitavamo elemente niza
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // za sve elemente
  for (int k = 0; k < n; k++) {
    // nalazimo poziciju najbližeg većeg prethodnika
    int p = najblizi_veci_prethodnik(a, k);
    // ispisujemo njegovu vrednost ili - ako ne postoji
    if (p != -1)
       cout << a[p] << endl;
    else
       cout << "-" << endl;
  }
```

```
  return 0;
}
```

Ova implementacija je neefikasna jer dolazi do višestrukog pozivanja funkcije za iste argumente. Stvar se može popraviti tako što primenimo tehniku dinamičkog programiranja (o njoj je bilo reči na kursu P2, a mnogo više reči o njoj će biti u nastavku ovog kursa) i pamtimo vrednosti rekurzivnih poziva u nizu.

```
// na poziciji k se nalazi vrednost najbližeg
// većeg prethodnika elementa a[k]
vector<int> najblizi_veci_prethodnik(n);

// početni element nema prethodnika
najblizi_veci_prethodnik[0] = -1;
// analiziramo redom naredne elemente
for (int k = 1; k < n; k++) {
  // pretragu počinjemo od neposrednog prethodnika tekućeg elementa
  int p = k-1;
  // dok god je prethodnik unutar niza, ali nije strogo veći od tekućeg
  while (p != -1 && a[p] <= a[k])
    // prelazimo na analizu njegovog prethodnika
    p = najblizi_veci_prethodnik[p];

  // za buduće potrebe pamtimo u nizu poziciju najbližeg većeg
  // prethodnika tekućeg elementa na poziciji k (tj. -1 ako ne postoji)
  najblizi_veci_prethodnik[k] = p;

  // ispisujemo vrednost najbližeg većeg prethodnika
  // ili - ako tekući element nema većih prethodnika
  if (p != -1)
    cout << a[p] << endl;
  else
    cout << "-" << endl;
}
```

Možda iznenađujuće, jer implementacija sadrži ugnežđene petlje, složenost najgoreg slučaja prethodne implementacije je linearna tj. $O(n)$, no to nije uopšte jednostavno dokazati. Ključni argument koji će nam pomoći i da uprostimo implementaciju i da lakše analiziramo složenost je to da tokom izvršavanja mnogi elementi niza prestaju da budu kandidati za najbliže veće prethodnike, pa se kroz njih nikada ne iterira tokom unutrašnje petlje `while`. Naime, kada se nakon nekog elementa x pojavi neki element y koji je veći od njega ili mu je jednak, tada element x prestaje da bude kandidat za najbližeg većeg prethodnika svih elemenata koji se javljaju u nizu iza y (jer će y uvek biti bliži prethodnik elementima iza sebe od x, koji će ako su strogo manji od x biti sigurno strogo manji i od y, jer je $x \leq y$). Zaista, ako se iza neke zgrade pojavi zgrada iste visine ili viša od nje ona će zakloniti prethodnu zgradu i nijedan kanap nadalje neće moći biti razvučen do te niže zgrade. Stoga se tokom petlje `while` iterira

kroz relativno mali broj kandidata. U svakom trenutku elementi koji su kandidati za najbliže manje predhodnike čine opadajući niz. Svaki naredni element eliminiše one tekuće kandidate koji su manji od njega ili su mu jednaki (svaka nova zgrada sakriva zgrade koje su niže od nje ili su joj iste visine). Prvi element u nizu potencijalnih kandidata koji je strogo veći od tekućeg elementa je ujedno njegov najbliži veći prethodnik. Ako takav element ne postoji, onda element nema većeg prethodnika.

Pošto je niz kandidata opadajući, kandidati koji bivaju eliminisani (tj. zaklonjeni) 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 (na steku možemo čuvati bilo pozicije tih kandidata, bilo njihove vrednosti). 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 $[i, j)$. Ovim dobijamo implementaciju veoma sličnu prethodnoj, i sa istom asimptotskom memorijskom složenošću najgoreg slučaja. Ipak, za mnoge ulaze, zauzeće memorije će biti manje (ako na steku čuvamo vrednosti, ne moramo čak ni da pamtimo ceo originalni niz).

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

using namespace std;

int main() {
  int n;
  cin >> n;
  // stek na kojem čuvamo kandidate za najbliže veće prethodnike
  stack<int> s;
  // obrađujemo sve elemente
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    // skidamo sa steka elemente koji prestaju da budu kandidati
    // jer ih je tekući element zaklonio
    while (!s.empty() && s.top() <= x)
        s.pop();
    if (s.empty())
       // ako na steku nema preostalih kandidata, tada
       // tekući element nema većih prethodnika
       cout << "-" << endl;
    else
       // element na vrhu steka je najbliži veći prethodnik
       // tekućem
       cout << s.top() << endl;

    // tekući element postaje kandidat za najbližeg većeg
```

```
    // prethodnika narednim elementima
    s.push(x);
  }
  return 0;
}
```

Prikažimo rad ovog algoritma na primeru niza `3 1 2 5 3 1 4`.

Na početku je stek prazan.

`Stek:`

Pošto je stek prazan, `3` nema manjih prethodnika. Nakon toga se on dodaje na stek.

`Stek: 3`

Pošto je element `1` manji od elementa `3` koji je na vrhu steka, element `3` je njegov najbliži veći prethodnik. Nakon toga se na stek dodaje `1`.

`Stek: 3 1`

Kada na red dođe element `2` sa steka se skida element `1` koji je manji od njega i pronalazi se da je najbliži veći prethodnik broja `2` broj `3`. Element `1` zaista prestaje da bude kandidat svim kasnijim elementima (zgrada visine `2` zaklanja zgradu visine `1`) i element `2` koji se postavlja na stek preuzima njegovu ulogu.

`Stek: 3 2`

Kada na red dođe element `5` sa steka se skidaju elementi `3` i `2`. Elementi `3` i `2` zaista prestaju da budu kandidati svim kasnijim elementima (zgrada visine `5` zaklanja zgrade visine `3` i `2`). Pošto je stek prazan, element `5` nema većih prethodnika. Element `5` se postavlja na stek.

`Stek: 5`

Pošto je element `3` manji od elementa `5` koji je na vrhu steka, element `5` je njegov najbliži veći prethodnik. Nakon toga se na stek dodaje `3`.

`Stek: 5 3`

Pošto je element `1` manji od elementa `3` koji je na vrhu steka, element `3` je njegov najbliži veći prethodnik. Nakon toga se na stek dodaje `1`.

`Stek: 5 3 1`

Kada na red dođe element `4` sa steka se skidaju elementi `1` i `3` koji su manji od njega i pronalazi se da je najbliži veći prethodnik broja `4` broj `5`. Elementi `3` i `1` zaista prestaju da budu kandidati svim kasnijim elementima (zgrada visine `4` zaklanja zgrade visine `1` i `3`) i element `4` koji se postavlja na stek preuzima njihovu ulogu.

Raspon akcija

Problem: Poznata je vrednost akcija tokom n dana. Definišimo da je rok važenja akcija nekog dana najduži period prethodnih uzastopnih dana u kojima je vrednost akcija manja ili jednaka vrednosti u tom danu. Odredi rok važenja akcija za svaki dan.

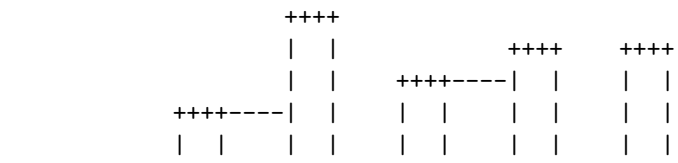
Ovaj zadatak je veoma sličan prethodnom, jedino što za svaki element ne treba da znamo vrednost, već poziciju njemu najbližeg prethodika koji je strogo veći od njega (tj. -1 ako takav prethodnik ne postoji). Sve elemente ćemo na početku učitati u niz, a na steku ćemo pamtiti pozicije (a ne vrednosti) preostalih kandidata.

```
stack<int> s;
for (int i = 0; i < n; i++) {
  while (!s.empty() && a[s.top()] <= a[i])
    s.pop();
  cout << (s.empty() ? i + 1 : i - s.top()) << endl;
  s.push(i);
}
```

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 (ako su svi elementi desno od pozicije i manji ili jednaki a_i), 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`.

Ovaj zadatak ima veoma sličnu interpretaciju kao onaj sa nalaženjem najbližih većih prethodnika, jedino što se kanapi sada šire nadesno. Na primer, ako je niz `1 4 2 3 3`, kanapi su raspoređeni kao na narednoj slici.



Primetimo da se ovde traže pozicije, a ne vrednosti sledbenika, ali to ništa značajno ne menja u postupku rešavanja, kao ni to da li se radi o većim ili manjim sledbenicima (ponovo je algoritam isti za bilo koju relaciju poretka).

Jedan način da se zadatak reši je da se koristi praktično isti postupak kao u prethodnom slučaju, ali da se niz obilazi zdesna na levo (u tom redosledu obilaska prethodnik je isto što i sledbenik u obilasku sleva nadesno).

Ipak, pokazaćemo i direktno rešenje nastalo modifikacijom tehnike koju smo videli prilikom traženja najbližih većih prethodnika, koje nam može biti korisno i u nekim drugim kontekstima. Invarijanta petlje će biti to da se na steku (u rastućem redosledu) nalaze pozicije svih elemenata čiji najbliži veći sledbenik još nije određen tj. oni elementi iza kojih još nije pronađen neki veći element. 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 (na osnovu invarijante znamo da se iza tih elemenata nije ranije pojavio element veći od njih, pa pošto je tekući element veći od njih upravo njima najbliži veći sledbenik). 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.

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

using namespace std;

int main() {
  // učitavamo elemente niza
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // pozicije najbližih većih sledbenika
  vector<int> p(n);

  // stek na kome se nalaze elementi čiji najbliži veći sledbenik
  // još nije obrađen
  stack<int> s;
  // obilazimo sve elemente niza sleva nadesno
  for (int i = 0; i < n; i++) {
    // skidamo sa steka sve element kojima je najbliži
    // veći sledbenik element na poziciji i
    while (!s.empty() && a[s.top()] < a[i]) {
      p[s.top()] = i;
```

```
      s.pop();
    }
    // elementu i još ne znamo poziciju najbližeg većeg sledbenika
    s.push(i);
  }

  // preostali elementi na steku nemaju većih sledbenika
  while (!s.empty()) {
    p[s.top()] = n;
    s.pop();
  }

  // ispisujemo sve pozicije
  for (int i = 0; i < n; i++)
    cout << p[i] << endl;

  return 0;
}
```

Razmotrimo rad algoritma na primeru niza `1 4 2 3 3`.

Stek je na početku prazan.

`Stek: Niz p: ? ? ? ? ?`

Posle toga obrađujemo element `1` i on nije najbliži veći sledbenik ni jednog elementa (jer je stek prazan). Pošto ni za element `1` još ne znamo najbližeg većeg sledbenika, postavljamo njegovu poziciju na stek.

`Stek: 0 Niz p: ? ? ? ? ?`

Posle toga obrađujemo element `4`. Pošto je on veći od elementa `1` čija se pozicija nalazi na vrhu steka, on je najbliži veći sledbenik elementu na poziciji `0`. Sa steka skidamo poziciju `0` i upisujemo poziciju `1` elementa `4`, jer za nju još ne znamo poziciju najbližeg većeg sledbenika.

`Stek: 1 Niz p: 1 ? ? ? ?`

Posle toga obrađujemo element `2`. Pošto je on manji od elementa `4` čija se pozicija nalazi na vrhu steka, još ne znamo rešenje za element `4`, pa njegova pozicija ostaje na steku. Na vrh steka dodajemo poziciju `2` elementa `2`, jer ni za nju ne znamo poziciju najbližeg većeg sledbenika.

`Stek: 1 2 Niz p: 1 ? ? ? ?`

Posle toga obrađujemo element `3`. Pošto je on veći od elementa `2` čija se pozicija nalazi na vrhu steka, on je najbliži veći sledbenik elementu na poziciji `2`. Tada sa steka skidamo poziciju `2`. Pošto je on manji od elementa `4` čija se pozicija nalazi na vrhu steka, još ne znamo rešenje za element `4`, pa njegova pozicija ostaje na steku. Na vrh steka dodajemo poziciju `3` elementa `3`, jer ni za nju ne znamo poziciju najbližeg većeg sledbenika.

`Stek: 1 3 Niz p: 1 ? 3 ? ?`

Posle toga obrađujemo drugi element `3`. Pošto je on jednak elementu `3` čija se pozicija nalazi na vrhu steka, još ne znamo rešenje za element `3`, pa njegova pozicija ostaje na steku. Na vrh steka dodajemo poziciju `4` elementa `3`, jer ni za nju ne znamo poziciju najbližeg većeg sledbenika.

`Stek: 1 3 4 Niz p: 1 ? 3 ? ?`

Pošto smo stigli do kraja niza, elementi koji su ostali na steku nemaju većih sledbenik i na te pozicije u nizu `p` upisujemo dužinu niza `5`.

`Stek: Niz p: 1 5 3 5 5`

Segmenti oivičeni maksimumima

Problem: Odrediti koliko u nizu koji sadrži sve različite elemente postoji segmenata (bar dvočlanih podnizova uzastopnih elemenata niza) u kojima su svi elementi untar segmenta strogo manji od elemenata na njihovim krajevima.

I ovom slučaju je, naravno, direktno rešenje prilično neefikasno. Bolje rešenje možemo dobiti tako što prilagodimo tehniku određivanja pozicija najbližih prethodnika strogo većih od datog elementa.

Za svaku poziciju j ćemo odrediti broj segmenata koji zadovoljavaju dati uslov, a kojima je desni kraj na poziciji j. Ukupan broj takvih segmenata će biti zbir brojeva segmenata za svako j.

Da bi segment oivičen maksimumima mogao da počne na poziciji i neophodno je da važi da je a_i maksimum svih elemenata na pozicijama u intervalu $[i, j)$. Pretpostavimo da znamo sve pozicije $i_1 < i_2 < ... < i_k$ koje zadovoljavaju takav uslov (znamo da su to jedini preostali kanidati za leve krajeve segmenata koji su maksimalni). Elementi na tim pozicijama su opadajući tj. važi $a_{i_1} > a_{i_2} > ... > a_{i_k}$. Pozicija $j-1$ sigurno je poslednja u nizu (jer je a_{j-1} maksimum segmenta određenog jednočlanim intervalom pozicija $[j-1, j)$). Dodatno, svi elementi između dve susedne pozicije su manji i od prve i od druge (jer bi u suprotnom poslednji od njih koji je veći od druge pozicije bio veći od svih elemenata iza sebe i morao bi da se nađe u nizu). Na primer, ako je prefiks niza ispred pozicije j niz $8, 4, 6, 3, 7, 2, 5$, karakterističi maksimumi su elementi $8, 7, 5$ (8 je maksimum celog niza, 7 je maksimum dela niza koji započinje, i isto važi i za 5, oni čine opadajući niz i između bilo koja dva od njih nalaze se elementi strogo manji od njih).

Ako za neku takvu poziciju i važi i da je $a_i < a_j$, onda interval $[i, j]$ sigurno zadovoljava uslov zadatka (svi elementi u unutrašnjosti su manji od elemenata na krajevima). Ako je $a_i > a_j$, to ne mora biti slučaj, međutim, analizom našeg niza karakterističnih maksimuma, možemo identifikovati tražene segmente. Analizu započinjemo zdesna i za svaki element a_i koji je manji od a_j znamo da smo pronašli segment koji zadovoljava uslove zadatka. Prvi maksimum zdesna koji

je veći od a_j takođe određuje segment koji zadovoljava uslove zadatka. Ako je to a_{j-1}, uslovi su trivijalno ispunjeni, jer između a_{j-1} i a_j nema drugih elemenata niza. Ako je to neki $a_{i_m} > a_j$ takav da je $a_{i_{m+1}} < a_j$, uslovi su opet ispunjeni, jer su svi elementi na pozicijama od i_{m+1} do $j-1$ manji od a_j (jer su manji ili jednaki $a_{i_{m+1}}$), što takođe važi i za elemente na pozicijama od $i_m + 1$ do $i_{m+1} - 1$, jer smo već ranije konstatovali da su svi elementi između dva susedna maksimuma strogo manja od oba od njih, pa su manja i od $a_{i_{m+1}}$ koji je strogo manji od a_j. Ako je $a_{i_m} > a_j$ i $a_{i_{m+1}} > a_j$, jasno je da je element $a_{i_{m+1}}$, između pozicija i_m i j, a da je veći od minimuma elemenata na tim pozicijama.

Dakle, pod pretpostavkom da je poznat niz karakterističnih maksimuma za prefiks niza koji se završava na poziciji j, možemo jednostavno da odredimo broj segmenata koji se završavaju na toj poziciji i koji zadovoljavaju uslove zadatka. Pokažimo i da je niz karakterističnih maksimuma moguće veoma lako održavati i inkrementalno ga ažurirati kada se prefiks produžava jednim po jednim elementom zdesna. Naime, ako prefiks proširujemo elementom a_j, tada ni jedan element a_i koji je ranije bio maksimum i manji je od a_j više nije maksimum (zato što prefiks sada na svom desnom kraju uključuje i element a_j koji ranije nije bio uključen). Svi raniji maksimumi a_i koji su veći od a_j ostaju da budu maksimumi. Na kraju, a_j je takođe maksimum.

Na osnovu svega rečenog, možemo formulisati jednostavan algoritam za rešavanje ovog zadatka. Primetimo da se niz karakterističnih maksimuma ponaša kao stek (elemente dodajemo na desni kraj i obrađujemo ih i uklanjamo sa tog desnog kraja). Prvi element niza postavljamo na stek. Nakon toga, obrađujemo jedan po jedan element niza a_j, od drugog pa do poslednjeg. Skidamo jedan po jedan element sa vrha steka koji su manji od a_j i za svaki od njih uvećavamo broj segmenata i to dok se stek ne isprazni ili dok se na vrhu steka ne nađe element veći od a_j. U ovom drugom slučaju broj segmenata uvećavamo za još jedan. Nakon toga element a_j postavljamo na vrh steka.

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

using namespace std;

int main() {
  int n;
  cin >> n;
  // ukupan broj segmenata oivičenih maksimumima
  int brojSegmenata = 0;
  // na steku se čuvaju svi elementi ai takvi da je ai maksimum
  // elemenata na pozicijama [i, j), gde je i pozicija elementa ai
  stack<int> s;
  for (int j = 0; j < n; j++) {
    int aj;
    cin >> aj;
    while (!s.empty() && s.top() < aj) {
```

```
      // ako je ai = s.top() i ako je i njegova pozicija,
      // tada je segment na pozicijama [i, j] oivičen maksimumima
      brojSegmenata++;
      s.pop();
    }
    if (!s.empty())
      // ako je ai = s.top() i ako je i njegova pozicija,
      // tada je segment na pozicijama [i, j] maksimumima
      brojSegmenata++;

    // svi preostali elementi na steku su i dalje maksimumi
    // segmenta [ai, j+1), pa treba da ostanu na steku
    // važi i da je aj maksimum segmenta [j, j+1),
    // pa ga treba dodati na stek
    s.push(aj);
  }
  cout << brojSegmenata << endl;
  return 0;
}
```

Prikažimo rad algoritma na narednom primeru niza `8 4 6 3 7 2 5`.

```
Stek:
Stek: 8
Stek: 8 4     - pronađen segment [8, 4]
Stek: 8       - pronađen segment [4, 6]
Stek: 8 6     - pronađen segment [8, 4, 6]
Stek: 8 6 3   - pronađen segment [6, 3]
Stek: 8 6     - pronađen segment [3, 7]
Stek: 8       - pronađen segment [6, 3, 7]
Stek: 8 7     - pronađen segment [8, 4, 6, 3, 7]
Stek: 8 7 2   - pronađen segment [7, 2]
Stek: 8 7     - pronađen segment [2, 5]
Stek: 8 7 5   - pronađen segment [7, 2, 5]
```

Najveći pravougaonik u histogramu

Problem: Niz brojeva predstavlja visine stubića u histogramu (svaki stubić je jedinične sirine). Odredi površinu najvećeg pravougaonika u tom histogramu.

Za svaki stubić u histogramu potrebno je da odredimo poziciju prvog stubića levo od njega koji je strogo manji od njega (ili poziciju -1 neposredno ispred početka, ako takav stubić ne postoji) i poziciju prvog stubića desno od njega koji je strogo manji od njega (ili poziciju n, neposredno iza kraja niza, ako takav stubić ne postoji). Ako te pozicije obeležimo sa l i d, onda možemo zaključiti da je najveća površina pravougaonika koji sadrži tekući stubić $(d-l-1)\cdot h[i]$ (taj pravougaonik je visine $h[i]$, a pod pretpostavkom da je svaki stubić jedinične širine, ukupna širina mu je $d-l-1$). Pošto su svi stubići od pozicije $l+1$

do $d-1$ visine veće ili jednake od $h[i]$, takav pravougaonik je moguće upisati histogram. Jasno je i da ne može da postoji pravougaonik koji bi bio viši od ovoga (jer je i-ti stubić visine $h[i]$ i viši pravougaonik bi njega prevazišao) niti da postoji pravougaonik koji bi bio širi od ovoga (jer čak i da postoje stubići na pozicijama l i d, oni su niži od $h[i]$ i pravougaonik visine $h[i]$ ne bi mogao da se proširi i upiše u njih).

Naivan način da se za svako i odrede l i d je da za svaki stubić i iznova puštamo petlje koje nalaze odgovarajuće ivične stubiće.

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

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

  int max = 0;
  for (int i = 0; i < n; i++) {
    int l = i;
    while (l >= 0 && h[l] >= h[i])
      l--;
    int d = i;
    while (d < n && h[d] >= h[i])
      d++;
    int P = (d - l - 1) * h[i];
    if (P > max)
      max = P;
  }
  cout << max << endl;

  return 0;
}
```

Ovaj algoritam je prilično neefikasan (složenost najgoreg slučaja mu je očigledno $O(n^2)$ i ona nastupa, na primer, kada su svi stubići jednake visine).

Već smo razmatrali probleme određivanja najbližeg manjeg prethodnika i najbližeg manjeg sledbenika za svaki element niza i videli smo da oba problema možemo rešiti u vremenu $O(n)$ (zapravo, tražili smo veće prethodnike i sledbenike, ali to ništa suštinski ne menja). Jedno moguće rešenje je da u jednom prolazu odredimo pozicije najbližih manjih prethodnika svakog elementa, u drugom pozicije najbližih većih sledbenika za svaki element niza i u trećem da na osnovu tih pozicija izračunamo maksimalne površine pravougaonika za svaki stubić. Međutim, pokazaćemo da se zahvaljujući sličnosti algoritama za određi-

vanje najbližeg manjeg prethodnika i najbližeg manjeg sledbenika sve može uraditi samo u jednom prolazu.

Na stek ćemo postavljati one stubiće za koje još ne znamo poziciju najbližeg manjeg sledbenika (i za koje još nismo odredili površinu maksimalnog pravougaonika koji određuju). Stubiće obilazimo s leva na desno i za svaki stubić h_d na koji naiđemo sa vrha steka redom skidamo i obrađujemo jedan po jedan stubić h koji je strogo veći od njega. Svakom od tih stubića h stubić h_d je najbliži manji sledbenik. Ako ispod h na steku postoji neki stubić on je najbliži manji ili jednak prethodnik stubiću h. Da bismo našli najbližeg strogo manjeg prethodnika potrebno je da nastavljamo da sa steka skidamo sve stubiće čija je visina jednaka h. Stubić h_l koji se nalazi na vrhu steka nakon toga je najbliži strogo manji prethodnik stubiću h. Ako takvog stubića nema, stubić h nema manjih prethodnika i histogram koji mu odgovara se može širiti skroz do levog kraja histograma (tada je $l = -1$). Površina maksimalnog pravougaonika koji uključuje i stubić h je $h \cdot (d - l - 1)$. Primetimo da se za stubiće jedanke visine stubiću h koji su skinuti sa steka ne računa površina pravougaonika - za tim nema potrebe jer je ona jednaka površini pravougaonika određenog stubićem h. Recimo i da smo umesto potrage za najmanjim strogo manjim prethodnikom mogli da površine računamo na osnovu najbližih manje-jednakih prethodnika. U tom slučaju ne bismo imali petlju u kojoj se sa steka skidaju svi stubići čija je visina jednaka h. Tada bi se za svaki stubić računala površina, ali ne bi površine pravougaonika svih bile maksimalne. Algoritam bi i u tom slučaju bio korektan, jer bi se maksimalna površina pravougaonika određenog susednim stubićima iste visine korektno izračunala prilikom skidanja poslednjeg od njih sa steka. Kada se stubići sa vrha steka obrade (kada se stek isprazni ili kada se na njegovom vrhu nađe stubić h_l koji je manji ili jednak od h_d), na vrh steka se stavlja h_d.

Naglasimo i da je po dolasku do kraja niza, potrebno još obraditi sve stubiće koji su ostali na vrhu steka (to će biti stubići za koje ne postoji strogo manji sledbenik). Njihova obrada teče po istom prinicipu kao i ranije, osim što se za vrednost pozicije d uzima n. Zato je u implementaciji poželjno objediniti tu završnu obradu sa prvom fazom algoritma.

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

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

  int max = 0;
  stack<int> s;
```

```
  for (int d = 0; d < n || !s.empty(); d++) {
    while (!s.empty() && (d == n || a[s.top()] > a[d])) {
      int h = a[s.top()];
      s.pop();
      while (!s.empty() && a[s.top()] == h)
          s.pop();
      int l = s.empty() ? -1 : s.top();
      int P = (d - l - 1) * h;
      if (P > max)
    max = P;
    }
    if (d < n)
      s.push(d);
  }
  cout << max << endl;
  return 0;
}
```

Prikažimo rad algoritma na jednom primeru.

```
3 5 5 8 6 4 9 2 4

Stek: 0                    3
Stek: 0 1                  3 5
Stek: 0 1 2                3 5 5
Stek: 0 1 2 3              3 5 5 8
6 na poziciji 4 je najbliži strogo manji sledbenik za 8
Stek: 0 1 2                3 5 5
5 na poziciji 2 je najbliži strogo manji prethodnik za 8
P = 8
Stek: 0 1 2 4              3 5 5 6
4 na poziciji 5 je najbliži strogo manji sledbenik za 6
Stek: 0 1 2                3 5 5
5 na poziciji 2 je najbliži strogo manji prethodnik za 6
P = 12
4 na poziciji 5 je najbliži strogo manji sledbenik za 5
Stek: 0 1                  3 5
Stek: 0                    3
3 na poziciji 0 je najbliži strogo manji pretodnik za 5
P = 20
Stek: 0 5                  3 4
Stek: 0 5 6                3 4 9
2 na poziciji 7 je najbliži strogo manji sledbenik za 9
Stek: 0 5                  3 4
4 na poziciji 5 je najbliži strogo manji prethodnik za 9
P = 9
2 na poziciji 7 je najbliži strogo manji prethodnik za 4
```

```
Stek: 0                    3
3 na poziciji 0 je najbliži strogo manji prethodnik za 4
P = 24
2 na poziciji 7 je najbliži strogo manji sledbenik za 3
Stek:
3 nema strogo manjeg prethodnika
P = 21
Stek: 7                    2
Stek: 7 8                  2 4
4 nema najbližeg strogo manjeg sledbenika
Stek: 7                    2
2 na poziciji 7 je najbliži strogo manji prethodnik za 4
P = 4
2 nema najbližeg strogo manjeg sledbenika
Stek:
2 nema najbližeg strogo manjeg prethodnika
P = 18


maxP = 24
```

Red pomoću dva steka

Problem: Implementiraj funkcionalnost reda korišćenjem dva steka.

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

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();
}

int main() {
   push(1);
   push(2);
   cout << top() << endl;
   pop();
   push(3);
   cout << top() << endl;
   pop();
   cout << top() << endl;
   pop();
   return 0;
}
```

Redovi

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 (pod pretpostavkom da red nije prazan)
- `back` - očitava element na kraju reda (pod pretpostavkom da red nije prazan)
- `empty` - proverava da li je red prazan
- `size` - vraća broj elemenata u redu

Prikažimo upotrebu reda kroz nekoliko interesantnih primera.

Nerekurzivni BFS

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

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

```
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);
    }
  }
}


int main() {
  bfs(0);
  return 0;
}
```

Maksimalni zbir segmenta dužine k

Problem: Učitava se n brojeva. Napiši program koji određuje segment uzastopnih k elemenata sa najvećim zbirom. Voditi računa o zauzeću memorije.

Najjednostavniji način bi bio da se svi elementi učitaju u niz i da se zatim određuju zbirovi segmenata dužine k. Zbirove, naravno, možemo računati inkrementalno. Dva susedna segmenta dužine k imaju zajedničke sve elemente osim prvog elementa levog i poslednjeg elementa desnog segmenta. Dakle, 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. Da bismo ovo mogli realizovati 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.

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

```cpp
using namespace std;

int main() {
  // broj elemenata niza i duzina segmenta
  int n, k;
  cin >> n >> k;

  // red u kojem u svakom trenutku cuvamo tekuci segment
  queue<double> q;

  // ucitavamo prvi segment duzine k, smestamo elemente u red i
  // racunamo mu zbir
  double zbir = 0.0;
  for (int i = 0; i < k; i++) {
    double x; cin >> x;
    q.push(x);
    zbir += x;
  }

  // trenutna maksimalna suma segmenta i indeks njenog pocetka
  int maxPocetak = 0;
  double maxZbir = zbir;

  for (int i = 1; i <= n-k; i++) {
    // ucitavamo naredni element
    double x; cin >> x;
    // azuriramo zbir
    zbir = zbir - q.front() + x;
    // menjamo "najstariji" element u redu
    q.pop(); q.push(x);
    // ako je potrebno, azuriramo maksimum
    if (zbir >= maxZbir) {
      maxZbir = zbir;
      maxPocetak = i;
    }
  }

  // ispisujemo pocetak poslednjeg segmenta sa maksimalnom sumom
  cout << maxPocetak << endl;

  return 0;
}
```

Maksimalna bijekcija

Problem: Dat je konačni skup A i funkcija $f : A \to A$. Pronaći maksimalnu kardinalnost skupa $S \subseteq A$, takva da je restrikcija f na S bijekcija.

Ovo je zapravo pravi algoritamski problem u kome su redovi samo pomoćno

sredstvo u implementaciji. 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 (do ovog će se slučaja uvek doći tokom rekurzivnog postupka - zašto?). 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 (kada bi se dva originala slikala u istu sliku, za neku sliku ne bi postojao original koji se u nju slika). Utvrdili smo, dakle, 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 (to su oni koji imaju ulazni stepen nula). 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.

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

using namespace std;

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

```
  // računamo ulazne stepene svih elemenata
  vector<int> ulazniStepen(n, 0);
  for (int i = 0; i < n; i++)
    ulazniStepen[f[i]]++;

  // u redu čuvamo one elemente čiji je ulazni stepen 0
  queue<int> q;
  for (int i = 0; i < n; i++)
    if (ulazniStepen[i] == 0)
      q.push(i);
  // broj preostalih elemenata
  int brojElemenata = n;
  while (!q.empty()) {
    // izbacujemo element koji je na redu (on ima ulazni stepen 0)
    int i = q.front(); q.pop();
    brojElemenata--;
    // ako nakon izbacivanja elementa i slika f[i] ima
    // ulazni stepen 0 ubacujemo je u red
    if (--ulazniStepen[f[i]] == 0)
      q.push(f[i]);
  }
  cout << brojElemenata << endl;
  return 0;
}
```

Način upotrebe reda u ovom rešenju je često koristan i pri rešavanju drugih problema. Kada god je potrebno obrađivati nekoliko elemenata koji ne mogu da budu obrađeni istovremeno, oni se mogu smestiti u red i zatim obrađivati jedan po jedan. Ako se tokom obrade pojavljuju novi elementi koje je potrebno obraditi, oni se dodaju na kraj reda. Obrada traje sve dok se red ne isprazni. Invarijanta je obično da red sadrži sve one elemente za koje je utvrđeno da moraju da budu obrađeni.

Red sa dva kraja

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:

- `push_front` - dodavanje elemenata na početak reda,
- `push_back` - dodavanje elemenata na kraj reda,
- `pop_front` - uklanjanje elemenata sa početka reda,
- `pop_back` - uklanjanje elemenata sa kraja reda,
- `front` - očitava element na početku reda (pod pretpostavkom da red nije prazan),

- `back` - očitava element na kraju reda (pod pretpostavkom da red nije prazan),
- `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. Više reči o implementaciji svih ovih struktura biće dato kasnije.

Prikažimo korišćenje reda sa dva kraja kroz nekoliko primera.

Ograničena istorija pregledača veba

Problem: 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 `-`.

```
#include <iostream>
#include <string>
#include <deque>

using namespace std;

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;
}
```

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 dan 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 $O(k \cdot (n - k))$ što je za $k \approx n/2$ jednako $O(n^2)$.

Zadatak možemo rešiti i u složenosti $O(n)$, ako upotrebimo sličnu tehniku čuvanja karakterističnih maksimuma (ili minimuma), kao u slučaju kada smo određivali najbližeg manjeg prethodnika.

Slično kada smo određivali maksimalni zbir segmenta dužine k, 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, lako možemo utvrditi da to neće teći tako jednostavno. Naime, 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. Dakle, moramo čuvati maksimum celog segmenta, zatim maksimum dela segmenta iza tog maksimuma, zatim maksimum dela segmenta iza tog maksimuma itd. Na primer, ako je segment $3, 8, 6, 5, 6, 3$ potrebno je da pamtimo vrednosti $8, 6, 6, 3$. 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 (moguće je i da ih nema), novi element se dodaje na kraj niza karakterističnih maksimuma, jer je on maksimum jednočlanog segmenta koji sam čini.

Struktura podataka kojom modelujemo niz karakterističnih maksimuma treba da omogući ispitivanje vrednosti elementa koji se nalazi na početku, uklanjanje tog elementa, ispitivanje vrednosti elemenata koji se nalaze na kraju, uklanjanje tih elemenata i dodavanje novog elementa na kraj. Sve te operacije su omogućene u konstantnom vremenu kada se koristi red sa dva kraja.

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

```
using namespace std;

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

  // red u kome cuvamo trenutni segment
  queue<int> segment;

  // red karakteristicnih maksimuma tekuceg segmenta
  deque<int> maksimumi;
  for (int i = 0; i < n; i++) {
    if (i >= k) {
      // Posle prvih k koraka pocinjemo da skracujemo
      // tekuci segment
      int prvi = segment.front();
      segment.pop();
      // Ako je element koji se izbacuje jednak prvom elementu
      // u redu maksimuma, on se izbacuje iz reda.
      if (maksimumi.front() == prvi)
         maksimumi.pop_front();
    }

    // dodajemo novi element
    int ai;
    cin >> ai;
    segment.push(ai);

    // Azurira se red maksimuma
    while(!maksimumi.empty() && ai > maksimumi.back())
      maksimumi.pop_back();
    maksimumi.push_back(ai);

    // nakon sto je prvih k elemenata ubaceno u segment,
    // pocinjemo da u svakom koraku ispisujemo maksimume
    // segmenata
    if (i >= k - 1)
      cout << maksimumi.front() << endl;
  }

  return 0;
}
```

Prikažimo rad algoritma na primeru niza `3 8 6 3 1 5 9 4 2 7 6 5` i vrednosti $k = 4$.

```
i     segment     karakteristični maksimumi      izlaz
0     3           3
1     3 8         8
2     3 8 6       8 6
3     3 8 6 3     8 6 3                          8
4     8 6 3 1     8 6 3 1                        8
5     6 3 1 5     6 5                            6
6     3 1 5 9     9                              9
7     1 5 9 4     9 4                            9
8     5 9 4 2     9 4 2                          9
9     9 4 2 7     9 7                            9
10    4 2 7 6     7 6                            7
11    2 7 6 5     7 6 5                          7
```

Iako se unutar petlje `for` nalazi petlja `while` složenost algoritma jeste linearna tj. $O(n)$. Naime, unutrašnja petlja `while` se ne može izvršiti veliki broj puta. Ako se u nekom koraku njeno telo izvrši veliki broj puta, tada će se puno elemenata izbaciti iz reda karakterističnih maksimuma i već u naredom će taj red biti slabo popunjen, što znači da će se tada telo moći izvršiti znatno manji broj puta. Ukupan broj izvršavanja tela petlje `while` je zapravo ograničen sa n jer se svaki element niza samo jednom može dodati i samo jednom može ukloniti iz reda. Stoga je ukupno vreme izvršavanja algoritma linearno tj. iznosi $O(n)$.

Redovi sa prioritetom

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 (pod pretpostavkom da red nije prazan)
- `empty` - proverava da li je red prazan
- `size` - vraća broj elemenata u redu

Operacije `push` i `pop` su obično složenosti $O(\log k)$, gde je k broj elemenata u redu, dok su ostale operacije složenosti $O(1)$.

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* (engl. heap sort). Naziv dolazi od strukture podataka hip (engl. heap) koja se koristi za implementaciju reda sa prioritetom. U pitanju je varijacija algoritma sortiranja selekcijom (engl. selection sort) u kojem se, podsetimo se, u svakom koraku najmanji element dovodi na početak niza. Određivanje minimuma preostalih elemenata vrši se klasičnim algoritmom određivanja minimuma niza (tj. njegovog odgovarajućeg sufiksa) 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 (obično je složenosti $O(\log k)$, 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.

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

int main() {
   // ovo je način da se u C++-u definiše red sa prioritetom u kome su
   // elementi poređani u opadajućem redosledu prioriteta (ovde, vrednosti)
   priority_queue<int, vector<int>, greater<int>> Q;

   // učitavamo sve elemente niza i ubacujemo ih u red
   int n;
   cin >> n;
   for (int i = 0; i < n; i++) {
     int ai;
     cin >> ai;
     Q.push(ai);
   }
   // vadimo jedan po jedan element iz reda i ispisujemo ga
   while (!Q.empty()) {
     cout << Q.top() << " ";
     Q.pop();
   }
   return 0;
}
```

Memorijska složenost ove implementacije je $O(n)$, jer se u redu čuva svih n elemenata, a vremenska složenost je $O(n \log n)$ jer se n puta izvode operacije složenosti $O(\log n)$ - podsetimo se, to što se broj elemenata smanjuje u svakoj narednoj operaciji izbacivanja elemenata ne utiče na asimptotsku složenost. Jednačina koja opisuje fazu umetanja, a i fazu izbacivanja je $T(n) = T(n-1) + O(\log n)$, a njeno rešenje je $O(n \log n)$.

U narednom poglavlju ćemo videti i kako se ovaj algoritam može direktno implementirati uz pomoć niza, bez korišćenja naprednih bibliotečkih struktura podataka.

k najmanjih učitanih brojeva

Problem: Napisati program koji omogućava određivanje k najmanjih od n učitanih brojeva unetih sa ulaza. Voditi računa o prostornoj i vremenskoj efikasnosti.

Jedan način bi bio da se učita niz i da se sortira, ali bi se tim pristupom nepotrebno trošilo i vreme i prostor (zamislite da je potrebno odrediti hiljadu najvećih od milion učitanih elemenata - potpuno nepotrebno bi se smeštalo 999000 elemenata i potpuno nepotrebno bi se tokom sortiranja određivao njihov međusobni redosled). 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. Dakle, potrebno je da imamo strukturu podataka u kojoj efikasno možemo da odredimo najveći element, da taj najveći element izbacimo i da u strukturu ubacimo proizvoljni element. Te operacije nam omogućava red sa prioritetom. Memorijska složenost ove implementacije je $O(k)$ jer se u redu čuva najviše k elemenata, a vremenska je $O(n \log k)$ jer se $O(n)$ puta vrše operacije složenosti $O(\log k)$ (jednačina je $T(n) = T(n-1)+O(\log k)$).

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

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;
}
```

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 (reći ćemo levoj) kolekciji čuvamo sve elemente koji su manji ili jednaki središnjem, a u drugoj (reći ćemo 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 (što ne dopuštamo), 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. Dakle, leva kolekcija treba da bude takva da lako možemo da pronađemo i izbacimo njen najveći element, a desna da bude takva da lako možemo da pronađemo i izbacimo njen najmanji element, pri čemu obe kolekcije moraju da podrže efikasno ubacivanje proizvoljnih elemenata. 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.

```
#include <iostream>
#include <queue>
#include <string>
#include <vector>
#include <functional>
using namespace std;

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();
    }
  }
}

int main() {
  while (true) {
    string s;
    if (!(cin >> s))
      break;
    if (s == "m")
      cout << medijana() << endl;
    else if (s == "d") {
      int x;
      cin >> x;
      dodaj(x);
    }
  }
  return 0;
}
```

Ukupno se vrši n dodavanja elemenata, a u svakom koraku se vrše operacije složenosti $O(\log k)$, gde je k trenutni broj elemenata umetnutnih u redove (u svakom redu će biti zapravo oko $k/2$ elemenata, pa će broj operacija biti $O(\log k/2)$, no to je isto što i $O(\log k)$). Otud je ukupna složenost $O(n \log n)$.

Silueta zgrada - skyline

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) {
  }
};
```

Silueta je deo-po-deo konstantna funkcija i određena je intervalima konstantnosti $(-\infty, x_0)$, $[x_0, x_1)$, $[x_1, x_2)$, ..., $[x_{n-1}, +\infty)$, 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.

$$\begin{array}{ccccccccccccc} & & 0 & & h_0 & & h_1 & & ... & & h_{n-2} & & 0 & \\ -\infty & & x_0 & & x_1 & & x_2 & ... & x_{n-2} & & x_{n-1} & & +\infty \end{array}$$

Podrazumevamo da su krajnje tačke $-\infty$ i $+\infty$ i da su vrednosti na tim intervalima jednake nuli. Dakle, deo-po-deo konstantna funkcija se može predstaviti pomoću n tačaka $x_0, ... x_{n-1}$ i $n-1$ vrednosti $h_0, ..., h_{n-2}$. Jednostavnosti radi mi ćemo ovakve funkcije predstavljati pomoću n uređenih parova (x_0, h_0), (x_1, h_1), ..., (x_{n-2}, h_{n-2}) i $(x_{n-1}, 0)$. Dakle, 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;
```

Postoji nekoliko načina da se ovaj zadatak reši. Osnovna induktivna konstrukcija podrazumeva da umemo da nađemo siluetu sve osim jedne zgrade i da na kraju umemo da tu zgradu uklopimo u siluetu ostalih zgrada. Da bismo zgradu uklopili u postojeću siluetu potrebno nam je linearno vreme (iako deo siluete u koji se zgrada integriše možemo možda uraditi binarnom pretragom i iako bismo umetanje novih promena u siluetu mogli vršiti u konstantnom vremenu ako bismo umesto vektora upotrebili listu, potrebno je da prođemo i proanaliziramo sve promene u postojećoj silueti u delu gde se umeće nova zgrada, a njih može da bude puno ako je zgrada široka). To znači da će rešenje biti složenosti $T(n) = T(n-1) + O(n)$, $T(1) = O(1)$, što daje složenost $O(n^2)$.

Jedno efikasnije rešenje zasnovano na dekompoziciji ćemo prikazati kada se budemo detaljnije bavili tom tehnikom konstrukcije algoritama.

Drugo efikasnije rešenje možemo postići ako zgrade obrađujemo sleva nadesno. Silueta se menja samo u tačkama u kojma počinje ili se završava neka zgrada. Stoga 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 (uključujući eventualno i tu tačku) a čiji se desni kraj ne nalazi levo od te tačke (uključujući eventualno i te tačke). Tu na scenu stupaju strukture podataka.

Potrebno je da održavamo strukturu podataka u koju ćemo ubacivati zgradu po zgradu kako nailaze (kada se naiđe na njihov levi kraj), izbacivati zgrade kako prolaze (kada se naiđe na njihov desni kraj) i u svakom trenutku moći efikasno da pronađemo maksimum trenutno ubačenih zgrada. Problem sa ovim zahtevima je to što je teško efikasno ih ostvariti istovremeno. Naime, 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 (jer zgrade ne bi bile uređene po tom kriterijumu). Sa druge strane, ako bismo zgrade čuvali nekako uređene po koordinatama krajeva, pronalaženje one sa maksimalnom visinom bi bilo neefikasno. Prvi pristup je ipak bolji, jer ne možemo nikako da se odreknemo mogućnosti efikasnog nalaženja maksimuma. 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 (red sa prioritetom nam daje veoma efikasno izbacivanje zgrade koja je najviša, ali ne i ostalih zgrada). Ključni trik, koji se veoma često može upotrebiti kod korišćenja redova sa prioritetom 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. Naime, 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. Međutim, za razliku od trenutka kada smo naišli na njen desni kraj, kada je njeno izbacivanje bilo komplikovano, u ovom trenutku se ona nalazi na vrhu reda i izbacivanje se može izvršiti veoma efikasno. Dakle, analiziraćemo i izbacivaćemo jednu po jednu najvišu zgradu sa vrha reda sve dok ne dođemo do zgrade koja se još nije završila.

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

  // sortiramo sve zgrade na osnovu pocetka
  struct PorediPocetak {
```

```
  bool operator() (const zgrada& z1, const zgrada& z2) {
    return z1.a < z2.a;
  }
};
sort(begin(zgrade), end(zgrade), PorediPocetak());

// pravimo vektor svih pocetnih i krajnjih tacaka zgrada i za svaku
// tacku belezimo da li je pocetna
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);
}
// sortiramo sve znacajne tacke na osnovu koordinate
sort(begin(tacke), end(tacke), [](auto p1, auto p2){
    return p1.first < p2.first;
  });


// cuvamo visine do sada obradjenih zgrada tako da veoma brzo mozemo
// da odredimo zgradu najvece visine - koristimo red sa prioritetom
struct PorediVisinu {
  bool operator() (const zgrada& z1, const zgrada& z2) {
    return z1.h < z2.h;
  }
};
priority_queue<zgrada, vector<zgrada>, PorediVisinu> pq;

// obilazimo sve znacajne tacke sleva nadesno
int z = 0;
for (auto p : tacke) {
  int x = p.first;
  int je_pocetak = p.second;

  // dodajemo u red sve zgrade koje pocinju na trenutnoj koodrinati
  if (je_pocetak)
    while (z < zgrade.size() && zgrade[z].a == x)
       pq.push(zgrade[z++]);

  // trazimo najvisu visinu do sada zapocete zgrade
  // eliminisemo zgrade koje su do sada zavrsene
  while (!pq.empty() && pq.top().b <= x)
    pq.pop();
  int h = !pq.empty() ? pq.top().h : 0;

  // integrisemo visinu najvise zgrade na trenutnoj koordinati u
  // tekucu siluetu
  dodajPromenu(silueta, x, h);
}
```

```
  // vracamo rezultat
  return silueta;
}
```

Ostaje još pitanje kako novu promenu integrisati u postojeću siluetu. To nije teško, ali je potrebno obratiti pažnju na nekoliko specijalnih slučajeva (koji uglavnom nastupaju usled toga što više zgrada mogu imati isti početak ili kraj). 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 (druga promena je tada višak). 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 (ako je nova visina veća o postojeće). 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));
}
```

Skupovi

Skup je osnovni matematički pojam, a ponekad u programiranju imamo potrebu da održavamo skup elemenata. Savremeni programski jezici obično pružaju bibliotečku podršku za to.

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, o čemu će više biti reči u narednom poglavlju), pa su im vremenske i prostorne karakteristike donekle različite.

Skupovi podržavaju sledeće osnovne operacije (za pregled svih operacija upućujemo čitaoca na dokumentaciju):

- `insert` - umeće novi element u skup (ako već postoji, operacija nema efekta). Kada se koristi `set` složenost umetanja je $O(\log k)$, gde je k broj elemenata u skupu, a kada se koristi `unordered_set`, složenost najgoreg slučaja je $O(k)$, dok je prosečna složenost $O(1)$, pri čemu je amortizovana složenost uzastopnog dodavanja većeg broja elemenata takođe $O(1)$. Naglasimo i da konstante kod složenosti $O(1)$ mogu biti relativno velike.
- `find` - proverava da li skup sadrži dati element i vraća iterator na njega ili `end` ako je odgovor negativan. Tako se provera pripadnosti elementa `e` skupu `s` može izvršiti sa `if (s.find(e) != s.end()) ...` Složenost najgoreg slučaja ove operacije ako se koristi `set` je $O(\log k)$, a ako se koristi `unordered_set` složenost najgoreg slučaja je $O(k)$, ali je prosečna složenost $O(1)$.
- `erase` - uklanja dati element iz skupa. Složenost je ista kao u prethodnom slučaju.

Prikažimo upotrebu skupova na nekoliko primera.

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

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

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 $O(n \log n)$, jer se n puta izvršava pretraga i umetanje u skup, a ta operacija ima složenost $O(\log k)$, gde je k trenutni broj elemenata u skupu. Ovim dobijamo sumu logaritama od 1 do n, a za nju smo već dokazali da je $\Theta(n \log n)$.

Naravno, zadatak je moguće rešiti i na druge načine. Jedan smo već videli - učitamo sve elemente u niz, sortiramo ga i onda proveravamo uzastopne elemente. Asimptotska složenost dolazi od sortiranja i takođe je $O(n \log n)$, ali ta varijanta može biti memorijski i vremenski malo efikasnija (naravno, samo za konstantni faktor), jer treba imati na umu da naprednije strukture podataka kakav je `set` često nose određenu cenu u ondnosu na klasične nizove.

Problem: Učitavaju se šifre proizvoda koji se nalaze u dva magacina. Napisati program koji određuje one koji se nalaze u oba.

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

int main() {
  int n1;
  cin >> n1;
  set<int> proizvodi1;
  for (int i = 0; i < n1; i++) {
     int x;
     cin >> x;
     proizvodi1.insert(x);
  }

  int n2;
  cin >> n2;
  set<int> proizvodi2;
  for (int i = 0; i < n2; i++) {
     int x;
     cin >> x;
     proizvodi2.insert(x);
  }

  for (int x : priozvodi1)
     if (proizvodi2.find(x) != proizvodi2.end())
        cout << x << endl;

  return 0;
}
```

Jezik C++ daje i funkcije koje izračunavaju presek, uniju i razliku dva skupa (`set_intersection`, `set_union`, `set_difference`).

```
   set<int> presek;
   set_intersection(begin(proizvodi1), end(proizvodi1)
                    begin(proizvodi2), end(proizvodi2),
                    inserter(presek, begin(presek)));
   for (int x : presek)
      cout << x << endl;
```

Mape

Mape (rečnici, konačna preslikavanja, asocijativni nizovi) 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. Na primer, ako mapa `ocene` preslikava imena učenika u njihove prosečne ocene, tada se ocena učenika `pera` može i pročitati i postaviti pomoću `ocene["pera"]`. Kada se koristi `map`, garantovana složenost ovog operatora je $O(\log k)$ 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. Tako se provera pripadnosti ključa `k` mapi `m` može izvršiti sa `if (m.find(k) != m.end()) ...`. 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.

Jezik dopušta i iteriranje kroz elemente mape. Na primer, kroz sve ocene se može proći sa sledećom petljom.

```
for (auto it : ocene)
  cout << it.first << " " << it.second << endl;
```

Primetimo da promenljiva `it` tokom iteracije uzima uređene parove (ključ, vrednost), pa se ključu pristupa preko polja `first`, a vrednosti preko polja `second`.

Problem: Organizuj strukturu podataka koja studentima pridružuje broj poena.

```
#include <iostream>
#include <map>
using namespace std;
int main() {
   map<string, int> poeni =
      {{"pera", 84}, {"ana", 92}, {"joca", 67}};
   cout << poeni["ana"] << endl;
   poeni["joca"] = 72;
```

```
   cout << poeni["joca"] << endl;
   poeni["ivana"] = 48;
   cout << poeni["ivana"] << endl;
   return 0;
}
```

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

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

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;
}
```

Naglasimo da se u slučaju kada je domen mali rečnici mogu realizovati i preko klasičnih nizova. Preuredimo prethodni program tako se broje slova unutra jedne reči koja se sastoji samo od malih slova. Tada rečnik možemo realizovati nizom od 26 elemenata i ključeve (mala slova) lako prevesti u ključeve (oduzimajući ASCII kôd karaktera `'a'`).

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

int main() {
  int frekvencije[26];
  string rec;
  cin >> rec;
  for (char c : rec)
    frekvencije[c - 'a']++;
  for (int i = 0; i < frekvencije.size(); i++)
    cout << (char)('a' + i) << " " << frekvencije[i] << endl;
  return 0;
}
```

Broj segmenata čiji su svi elementi različiti

Prikažimo kako se mape mogu upotrebiti i u rešenjima složenijih algoritamskih zadataka.

Problem: Dat je niz celih brojeva. Odrediti ukupan broj segmenata čiji su svi elementi različiti.

Jedno prilično elegantno rešenje se zasniva na tome da za svaku poziciju analiziramo sve segmente kojima je kraj upravo na toj poziciji, a kojima su svi elementi različiti. Ako su svi elementi nekog segmenta različiti, onda su i svim njegovim sufiksima svi elementi različiti. Ako neki segment sadrži duplikate, onda duplikate sadrže i svi segmenti kojima je on sufiks. Zato je dovoljno da pronađemo najduži mogući segment koji se završava na poziciji $kraj$ i kojem su svi elementi različiti i znaćemo da su svi segmenti sa različitim elementima koji se završavaju na poziciji $kraj$ tačno svi njegovi sufiksi. Ako je to segment određen pozicijama iz intervala $[pocetak, kraj]$ onda će takvi biti i segmenti određeni intervalima pozicija $[pocetak + 1, kraj]$, ..., $[kraj - 1, kraj]$. Njih je ukupno $kraj - pocetak$.

Ostaje pitanje kako za datu poziciju $kraj$ odrediti poziciju $pocetak$. Pokušajmo da primenimo induktivno-rekurzivnu konstrukciju. Za $kraj = 0$ važi da je $pocetak = 0$, jer su svi elementi jednočlanog segmenta koji čini samo element na poziciji 0 različiti i on je najduži takav segment koji se zavšava na poziciji 0. Pretpostavimo da je $kraj > 0$ i da već znamo rešenje za prethodnu poziciju kraja, tj. pretpostavimo da znamo da interval pozicija $[pocetak, kraj - 1]$ određuje najduži segment kome su svi elementi različiti i koji se završava na poziciji $kraj - 1$.

Ako se element na poziciji $kraj$ ne sadrži u tom segmentu, onda je segment $[pocetak, kraj]$ naš traženi. Ako se sadrži, onda je on sigurno jedini duplikat u segmentu $[pocetak, kraj]$. Ako pretpostavimo da se element na poziciji $kraj$ u tom segmentu javlja i na poziciji p, tada je segment koji tražimo $[p + 1, kraj]$, zato što svi segmenti koji počinju od pozicije $pocetak$, pa sve do pozicije p sadrže isti taj duplikat. Segment $[p + 1, kraj]$ ne sadrži duplikate i najduži je takav segment, tako da u toj situaciji $pocetak$ treba postaviti na vrednost $p + 1$.

Na kraju, ostaje pitanje kako utvrditi da li se a_{kraj} javlja u segmentu $[pocetak, kraj - 1]$ i ako se javlja, kako odrediti poziciju p na kojoj se javlja. Direktno rešenje podrazumeva linearnu pretragu segmenta prilikom svakog proširenja niza, što bi znatno degradiralo složenost celog algoritma. Bolje rešenje je da se čuva asocijativni niz (mapa, rečnik) u kojem se elementi niza iz segmenta pozicija $[pocetak, kraj - 1]$ preslikavaju u njihove pozicije. Tada se jednostavnom pretragom mape tj. rečnika (čija je složenost konstantna ili najviše logaritamska) utvrđuje da li se novi krajnji element javlja u prethodnom segmentu i na isti način se određuje i njegova pozicija. Recimo i da se prilikom pomeranja početka na poziciju $p + 1$ segment skraćuje, što treba da se oslika i u mapi - zato je tada potrebno iz mape ukloniti sve elemente koji se javljaju u nizu, na pozicijama od $pocetak$, pa zaključno sa p.

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

```cpp
using namespace std;

int main() {
  // ucitavamo dati niz brojeva
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ukupan broj segmenata niza ciji su svi elementi razliciti
  int broj = 0;

  // za svaku poziciju kraj zelimo da pronadjemo najduzi segment
  // oblika [pocetak, kraj] koji ima sve razlicite elemente

  // za svaki element u tekucem segmentu [pocetak, kraj] pamtimo
  // poziciju na kojoj se pojavljuje
  unordered_map<int, int> prethodno_pojavljivanje;

  int pocetak = 0;
  for (int kraj = 0; kraj < n; kraj++) {
    if (prethodno_pojavljivanje.find(a[kraj]) != prethodno_pojavljivanje.end()) {
      // nijedan segment koji se zavrsava na poziciji kraj, a pocinje
      // pre ranijeg pojavljivanja elementa a[kraj] ne moze da ima sve
      // razlicite elemente, pa zato razmatramo samo segmente koji se
      // zavrsavaju na poziciji kraj i pocinju iza pozicije tog
      // prethodnog pojavljivanja - najduzi takav pocinje na prvoj
      // poziciji iza te pozicije
      int novi_pocetak = prethodno_pojavljivanje[a[kraj]] + 1;
      // brisemo iz segmenta sve elemente od starog do ispred novog pocetka
      // i mapu uskladjujemo sa time
      for (int i = pocetak; i < novi_pocetak; i++)
        prethodno_pojavljivanje.erase(a[i]);
      // pomeramo pocetak
      pocetak = novi_pocetak;
    }
    // prosirujemo segment elementom a[kraj], pa pamtimo poziciju
    // njegovog pojavljivanja
    prethodno_pojavljivanje[a[kraj]] = kraj;

    // [pocetak, kraj] sadrzi sve razlicite elemente i on je najduzi
    // takav koji se zavrsava na poziciji kraj
    // sigurno su takvi i [pocetak+1, kraj], ..., [kraj-1, kraj]
    // njih ima (kraj - pocetak) i taj broj dodajemo na ukupan broj
    // trazenih segmenata
    broj += kraj - pocetak;
  }
```

```
  // ispisujemo ukupan broj pronadjenih segmenata
  cout << broj << endl;

  return 0;
}
```

Broj segmenata datog zbira

Problem: Napiši program koji za dati niz celih brojeva određuje sve neprazne segmente uzastopnih elemenata niza čiji je zbir jednak datom broju.

Zbirovima segmenata niza celih brojeva smo se već bavili kada smo tražili onaj najvećeg zbira, ali je sada zadatak malo drugačiji.

Direktno rešenje grubom silom podrazumevalo bi da se provere svi segmenti uzastopnih elemenata, da se za svaki izračuna zbir i da se proveri da li je taj zbir jednak traženom. Svi segmenti se mogu nabrojati ugnežđenim petljama. Složenost ovog pristupa je kubna tj. $O(n^3)$ nu odnosu na dimenziju n (postoji kvadratni broj segmenata i za sabiranje elemenata svakog od njih potrebno je linearno vreme).

Prethodni algoritam se može unaprediti ako se zbirovi računaju inkrementalno čime se složenost celog algoritma redukuje na kvadratnu tj. $O(n^2)$.

Elegantan i često primenjivan način da se dobiji zbirovi svih segmenata uzastopnih brojeva je da se izračunaju zbirovi prefiksa i da se zbir elemenata segmenta $[i, j]$ izrazi kao razlika zbira elemenata segmenta $[0, j]$ i zbira elemenata segmenta $[0, i-1]$. Dakle, u pomoćni niz b na svaku poziciju k možemo smestiti zbir prvih k elemenata niza (ovo opet možemo uraditi inkrementaln). Prvi element niza b je nula i on ima jedan element više od niza a. Tako se zbir elemenata niza a iz segmenta $[i, j]$ uvek određuje kao $b_{j+1} - b_i$. Zbirovi prefiksa (parcijalne sume niza) se mogu u jeziku C++ odrediti i bibliotečkom funkcijom `partial_sum` koja prima dva iteratora koji ograničavaju deo niza (ili vektora) čije se parcijalne sume izračunavaju i iterator koji ukazuje na početak dela niza (ili vektora) u koji se parcijalne sume upisuju. Primetimo da bismo zbirove prefiksa mogli smestiti i u sam niz a, ako je memorija kritičan resurs. Ako proveravamo svaki par elemenata $i < j$, ponovo dobijamo algoritam kvadratne složenosti.

Prethodna ideja nam daje mogućnost da stignemo do efikasnijeg rešenja. Problem možemo formulisati i ovako. Za svaki zbir b_{j+1} prefiksa $[0, j]$ potrebno je da pronađemo da li postoji zbir b_i prefiksa $[0, i)$ za $i < j$ takva da je $b_{j+1} - b_i = z$, gde je z traženi zbir, tj. da se proveri da li se među zbirovima prethodnih prefiksa nalazi vrednost $b_i = b_{j+1} - z$. Ako se ta pretraga vrši linearno, dolazimo do implementacije veoma slične prethodnoj, koja ispituje svaki par elemenata $i < j$. Pošto među elementima niza može biti i negativnih, zbirovi prefiksa nisu sortirani i ne možemo primeniti ni binarnu pretragu. Ostaje nam, međutim, mogućnost da u nekoj strukturi podataka koja omogućava efikasno pretraživanje čuvamo sve zbirove prefiksa za indekse $i < j$. Ako algoritam organizu-

jemo tako da j uvećavamo od 0 do $n-1$, tada se na kraju svakog koraka u tu strukturu može dodati i zbir tekućeg segmenta (b_{j+1}). Struktura treba da realizuje pretragu po ključu, tako da je najbolje upotrebiti asocijativno preslikavanje, odnosno mapu tj. rečnik. Pošto se u zadatku traži samo određivanje broja segmenata sa datim zbirom, ključevi mogu biti zbirovi prefiksa, a vrednost pridružena svakom ključu može biti broj pojavljivanja prefiksa sa tim zbirom. Da se tražila samo provera da li postoji segment sa datim zbirom, mogli smo umesto preslikavanja čuvati samo skup ranije viđenih vrednosti zbirova prefiksa, a da su se eksplicitno tražili svi prefiksi, onda bismo svaki ključ preslikavali u niz vrednosti i takvih da je b_i jednako tom ključu (mogla bi se upotrebiti i multimapa o čemu će više reči biti kasnije). Ako računamo da će pretraga biti realizovana u $O(\log n)$ (što je najčešće slučaj ako se koriste strukture podataka zasnovane na binarnim stablima, kao što je u slučaju `map`), tada će ukupna složenost ove implementacije biti $O(n \log n)$, što je znatno efikasnije nego prethodne implementacije. Napomenimo da smo dobitak na efikasnosti platili dodatnom memorijom koju smo angažovali, međutim, ako je memorija kritičan resurs u ovom scenariju nije neophodno pamtiti polazni niz, tako da memorijska složenost neće biti značajno povećana.

```
int brojSegmenataDatogZbira(const vector<int>& a, int trazeniZbir) {
  // ucitavamo trazeni zbir
  int trazeniZbir;
  cin >> trazeniZbir;

  // zbir prefiksa
  int zbirPrefiksa = 0;

  // broj segmenata sa trazenim zbirom
  int broj = 0;

  // broj pojavljivanja svakog vidjenog zbira prefiksa
  map<int, int> zbiroviPrefiksa;
  // zbir pocetnog praznog prefiksa je 0 i on se za sada pojavio
  // jednom
  zbiroviPrefiksa[0] = 1;

  // ucitavamo elemente niza niz
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    // prosirujemo prefiks tekucim elementom
    zbirPrefiksa += x;

    // trazimo broj pojavljivanja vrednosti zbirPrefiksa - trazeniZbir
    // i azuriramo broj pronadjenih segmenata
    auto it = zbiroviPrefiksa.find(zbirPrefiksa - trazeniZbir);
```

```
    if (it != zbiroviPrefiksa.end())
      broj += it->second;

    // povecavamo broj pojavljivanja trenutnog zbira
    zbiroviPrefiksa[zbirPrefiksa]++;
  }

  return broj;
}
```

Još jedan način da se efikasno pretražuju zbirovi prefiksa je da se umesto mape svi oni smeste u niz, da se onda taj niz sortira i da se primeni binarna pretraga. Međutim, ovde treba biti obazriv, jer se nakon sortiranja na osnovu zbira izgubi originalni poredak prefiksa. Zbir segmenta se može dobiti samo ako se od zbira dužeg prefiksa oduzme zbir nekog kraćeg (nikako obratno). Stoga je potrebno da uz svaki zbir prefiksa pamtimo i dužinu tog prefiksa. Nakon sortiranja, obrađujemo jedan po jedan prefiks polaznog niza i ako je zbir tekućeg prefiksa z_1 binarnom pretragom pronalazimo sve one prefikse čiji je zbir z_2 takve da je $z_1 - z_2 = z$, gde je z traženi zbir, a koji su kraći od segmenta čiji je zbir z_1 (više reči o primenama binarne pretrage biće dato kasnije).

Multiskupovi i multimape

U skupu se svaki element pojavljuje najviše jednom. Multiskup je struktura podataka u kojoj se elementi mogu pojavljivati i više puta. Multiskup, dakle, odgovara mapi koja slika ključeve u njihov broj pojavljivanja (i tako se interno može implementirati).

U jeziku C++ multiskupovi se reprezentuju objektima klase `multiset<T>`, gde je `T` tip elemenata multiskupa. Na raspolaganju su nam sledeće operacije.

- `insert` umeće dati element u multiskup.
- `erase` briše element iz multiskupa. Ako je argument iterator, briše se element na koji taj iterator pokazuje, a ako je argument vrednost, brišu se sva pojavljivanja te vrednosti.
- `count` izračunava broj pojavljivanja datog elementa u multiskupu.

Ilustrujmo upotrebu multiskupova na par primera.

Problem: Sortirati sve reči koje se čitaju sa ulaza. Ako se neka reč pojavila više puta, prikazati je više puta.

Jedan od načina je da se sve reči umetnu u multiskup i da se zatim ispišu redom svi elementi multiskupa. Iako je ovaj pristup malo sporiji od korišćenja običnog niza niski i bibliotečkog sortiranja, asimptotska složenost mu je i dalje $O(n \log n)$.

```cpp
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main() {
   multiset<string> reci;

   string rec;
   while (cin >> rec)
     reci.insert(rec);

   for (auto rec : reci)
     cout << rec << endl;

   return 0;
}
```

Problem: Sa ulaza se unosi broj n i zatim n reči. Nakon toga se unosi broj k i k reči. Napisati program za svaku od tih k učitanih reči određuje koliko se puta pojavila među prvih n reči.

Pored elementarnog rešenja koje bi smestilo sve reči u niz i sortiralo taj niz, jedan način koji se često koristi je da se upotrebi mapa kojom se svaka reč preslikava u njen broj pojavljivanja. U narednom rešenju umesto mape, koristimo multiskup.

```cpp
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main() {
   // učitavamo reči i smeštamo ih u multiskup
   multiset<string> reci;
   int n;
   cin >> n;
   for (int i = 0; i < n; i++) {
      string s;
      cin >> s;
      reci.insert(s);
   }

   // za k reči očitavamo i prijavljujemo broj pojavljivanja
   int k;
   cin >> k;
   for (int i = 0; i < k; i++) {
      string s;
      cin >> s;
      cout << s << ": " << reci.count(s) << endl;
   }
```

```
  return 0;
}
```

Multimape dopuštaju da se isti ključ preslikava u više od jedne vrednosti. Jedan način da se realizuju je preslikavanje ključeva u kolekciju (niz, vektor) vrednosti.

U jeziku C++ se multimape direktno predstavljaju objektima klase `multimap<K, V>`, gde je `K` tip ključeva, a `V` tip vrednosti elemenata. Na raspolaganju imamo sledeće operacije.

- `insert` pridružuje novu vrednost ključu. Ključ i vrednost se zadaju kao argument i to kao par (vrednost tipa `pair<K, V>`). Ova operacija ima garantovanu složenost najgoreg slučaja $O(\log k)$, gde je k broj trenutnih elemenata u multimapi.
- `find` vraća iterator koji ukazuje na jednu vrednost pridruženu datom ključu ili `end`, ako ključu nije pridružena ni jedna vrednost. I ova operacija ima garantovanu složenost $O(\log k)$.
- `equal_range` vraća par iteratora koji ograničavaju deo multimape koji sadrži sve vrednosti pridružene datom ključu. I ova operacija ima garantovanu složenost $O(\log k)$.

Problem: Napisati program koji učitava imena gradova i zemalja u kojima se nalaze. Napiši program koji ispisuje te podatke sortirane po zemljama, a zatim posebno ispisuje sve gradove iz Srbije.

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

int main() {
  multimap<string, string> gradovi;

  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    string grad, zemlja;
    cin >> grad >> zemlja;
    gradovi.insert(make_pair(zemlja, grad));
  }

  for (auto it : gradovi)
    cout << it.first << " " << it.second << endl;

  auto gradovi_srbije = gradovi.equal_range("Srbija");
  for (auto it = gradovi_srbije.first; it != gradovi_srbije.second; it++)
    cout << it->second << endl;
```

```
  return 0;
}
```

Osmišljavanje apstraktnih struktura podataka

Problem: Opisati realizaciju apstraktne strukture podataka koja podržava sledeće operacije:

- `umetni(x)`: umetni ključ `x` u strukturu podataka, ako ga tamo već nema;
- `obriši(x)`: obriši ključ `x` iz strukture podataka (ako ga ima);
- `naredni(x)`: pronađi najmanji ključ u strukturi podataka koji je strogo veći od `x`. Izvršavanje svake od ovih operacija treba da ima vremensku složenost $O(\log k)$ u najgorem slučaju, gde je k broj elemenata u strukturi podataka.

Pogodna struktura podataka za realizaciju ovog tipa je skup i implementacija skupa u jeziku C++ podržava sve tri operacije. Umetanje se vrši metodom `insert`, brisanje metodom `erase`, dok se nalaženje prvog strogo većeg elementa može realizovati metodom `upper_bound`. Metodom `lower_bound` može se pronaći prvi element koji je veći ili jednak datom. Obe metode vraćaju iterator koji ukazuje iza kraja skupa, ako traženi element ne postoji.

Implementacija može biti sledeća.

```
set<int> s;

void umetni(int x) {
  s.insert(x);
}

void obrisi(int x) {
  s.erase(x);
}

bool naredni(int x, int& y) {
  auto it = s.upper_bound(x);
  if (it == s.end()) return false;
  y = *it;
  return true;
}
```

Problem: Konstruisati apstraktnu strukturu podataka za čuvanje preslikavanja ključeva u vrednosti. Struktura podataka treba da podržava sledeće operacije:

- `vrednost(x)`: odrediti vrednost pridruženu elementu sa ključem `x` (nula, ako ključ `x` nema pridruženu vrednost);
- `umetni(x, y)`: ključu `x` pridruži vrednost `y`;
- `obriši(x)`: obriši element sa ključem `x`;

- `uvećaj(x, y)`: vrednost pridruženu ključu `x` uvećaj za `y`;
- `uvećajSve(y)`: vrednosti pridružene svim ključevima uvećaj za `y`.

Vreme izvršavanja svake od ovih operacija u najgorem slučaju treba da bude $O(\log k)$, gde je k broj ključeva kojima su pridružene vrednosti.

Sve operacije osim poslednje direktno su podržane mapama. Da bi se realizovala operacija `uvećajSve(y)` možemo da održavamo promenljivu `uvećanje` koja će sadržati zbir svih vrednosti uvećanja, tj. zbir svih argumenata funkcije `uvećajSve`. Prema tome, `uvećajSve(y)` prosto dodaje vrednost `y` promenljivoj `uvećanje`, što se realizuje u složenosti $O(1)$. Da bi struktura funkcionisala ispravno, funkcija `vrednost(x)` treba da vrati zbir vrednosti pridruženoj ključu `x` i promenljive `uvećanje`. Funkcija `umetni(x, y)` ključu `x` treba da dodeli razliku vrednosti `y` i promenljive `uvećanje`.

Implementacija može biti sledeća.

```
map<int, int> m;
int uvecanje = 0;

int vrednost(int x) {
   auto it = m.find(x);
   if (it == m.end()) return 0;
   return *it + uvecanje;
}

void umetni(int x, int y) {
   m[x] = y - uvecanje;
}

void obrisi(int x) {
   m.erase(x);
}

void uvecaj(int x, int y) {
   m[x] += y;
}

int uvecajSve(int y) {
   uvecanje += y;
}
```

Problem: Dizajniraj apstraktnu strukturu podataka koja podržava naredne operacije.

- `umetni(x)`: umetanje se izvršava i ako je broj `x` već jednom prethodno umetnut u strukturu podataka; drugim rečima, struktura podataka treba da pamti duplikate;
- `y = ukloni()`: ukloni proizvoljan elemenat iz strukture podataka i dodeli ga promenljivoj y. Ako postoji više kopija istog elementa, uklanja se samo

jedna od njih.

Ovaj apstraktni tip podataka može se nazvati *skladište* (engl. pool). On se može iskoristiti za smeštanje poslova. Novi poslovi se generišu i umeću u skladište, a kad neki radnik postane raspoloživ, uklanja se neki posao. Sve operacije treba da se izvršavaju za vreme $O(1)$.

S obzirom na to da redosled izbacivanja elemenata nije preciziran postoji zaista mnogo načina da se ovakva struktura realizuje. Na primer, moguće je upotrebiti red ili stek.

```
queue<int> q;

void umetni(int x) {
   q.push(x);
}

int ukloni() {
   int x = q.top();
   q.pop();
   return x;
}
```

Problem: Kako se prethodna apstraktna struktura podataka može prilagoditi tako da se proizvoljan elemenat može pojaviti najviše jednom u strukturi podataka? Umetanje mora da bude praćeno proverom postojanja duplikata. Realizovati iste operacije kao u prethodnom slučaju, ali sa proverom postojanja duplikata. Koja je složenost izvođenja operacija?

Ako ne znamo raspon iz kojeg dolaze elementi, uz red možemo čuvati i skup elemenata koji se trenutno nalaze u strukturi podataka.

```
queue<int> red;
set<int> elementi;

void umetni(int x) {
  if (elementi.find(x) == elementi.end()) {
    red.push(x);
    elementi.insert(x);
  }
}

int ukloni() {
  int x = red.top();
  elementi.erase(x);
  red.pop();
  return x;
}
```

Složenost operacija je $O(\log n)$. Umesto `set` možemo koristiti i `unordered_set` uz prosečnu složenost $O(1)$ (ali složenost najgoreg slučaja $O(n)$).

Ako se unapred zna da će svi elementi biti iz nekog intervala $[0, n]$, onda umesto pomoću bibliotečkih klasa skup možemo realizovati pomoću niza logičkih vrednosti.

Kada se ne bi zahtevalo da se elementi uklanjaju u redosledu njihovog umetanja, tada bi se red mogao izbaciti i uvek izbacivati prvi (najmanji) element iz skupa.

```
int ukloni() {
   int x = *elementi.begin();
   elementi.erase(elementi.begin());
   return x;
}
```

Problem: Kako se prethodna apstraktna struktura podataka može prilagoditi tako da se svakom umetnutom elementu može pridružiti neka vrednost? Potrebno je realizovati sledeće operacije.

- `umetni(x, y)` - ključu `x` pridružuje se vrednost `y`. Ako je tom ključu vrednost dodeljena i ranije, ona se zanemaruje.
- `y = ukloni()` - uklanja i vraća proizvoljnu vrednost pridruženu nekom ključu.
- `nađi(x)` - vraća vrednost pridruženu nekom ključu.

Rešenje je slično kao prethodno, ali umesto skupa moramo upotrebiti mapu. Ako želimo da redosled uklanjanja odgovara redosledu umetanja, održavaćemo i red elemenata.

```
queue<int> red;
map<int, int> elementi;

void umetni(int x, int y) {
  if (elementi.find(x) == elementi.end()) {
    red.push(x);
    elementi[x] = y;
  }
}

int ukloni() {
  int x = red.front();
  int y = elementi[x];
  elementi.erase(x);
  red.pop();
  return y;
}

bool nadji(int x, int& y) {
  auto it = elementi.find(x);
  if (it == elementi.end())
    return false;
  y = it->second;
```

```
  return true;
}
```

Složenost svih operacija je $O(\log n)$. I u ovom slučaju je moguće upotrebiti `unordered_map`. Ako je raspon ključeva iz nekog intervala $[0, n]$, tada se mapa može realizovati pomoću običnog niza.

Ako redosled uklanjanja nije bitan, možemo izbaciti red i uvek brisati prvi element iz mape (onaj sa najmanjom vrednošću ključa).

```
int ukloni() {
   int x = elementi.begin()->first;
   int y = elementi.begin()->second;
   elementi.erase(elementi.begin());
   return y;
}
```

Kada je mapa realizovana pomoću niza, tada moramo održavati i neku listu ubačenih ključeva (red, stek, ...).

Problem: Neka je $S = \{s_1, s_2, \ldots, s_m\}$ vrlo veliki skup, izdeljen u k disjunktnih blokova. Kreirati apstraktnu stukturu podataka koja omogućava rad sa malim podskupovima T skupa S, i to sledeće operacije nad T:

- `umetni(si)` - umeće s_i u T
- `obrisi(si)` - briše s_i iz T
- `obrisiSveIzBloka(j)` - briše iz T sve elemente koji pripadaju bloku j.

Složenost svake od ovih operacija treba da bude $O(\log n)$, gde je n tekući broj elemenata u T. Pretpostaviti da su m i k veoma veliki brojevi, a da je n manji od njih.

Jednostavnosti radi pretpostavimo da je S skup celih brojeva. Možemo definisati mapu kojom se svakom elementu pridružuje broj bloka kom pripada.

```
map<int, int> blok;
```

Podskup T onda možemo predstaviti mapom kojom se svaki redni broj bloka slika u skup elemenata koji pripadaju tom bloku i podskupu T (umesto mape možemo koristiti i niz dimenzije k, ali pošto je n dosta manje od k mapom se štedi memorija, jer neće svi blokovi imati elemente u T).

```
map<int, set<int>> podskup;
```

Operacije je onda relativno jednostavno implementirati.

```
void umetni(int x) {
  int b = blok[x];
  podskup[b].insert(x);
}
```

```
void obrisi(int x) {
  int b = blok[x];
  podskup[b].erase(x);
}

void obrisiSveIzBloka(int b) {
  podskup.erase(b);
}
```

Čas 5.1, 5.2, 5.3 - implementacija struktura podataka

U poglavlju o korišćenja struktura podataka upoznali smo različite strukture podataka. U ovom poglavlju ćemo ih razvrstati po tome kako su implementirane.

- U grupu sekvencijalnih struktura podataka (kontejnera) spadaju `array`, `vector`, `list`, `forward_list` i `deque`.
- U grupu adaptora kontejnera spadaju `stack`, `queue` i `priority_queue`.
- U grupu asocijativnih kontejnera spadaju `set`, `multiset`, `map` i `multimap`.
- U grupu neuređenih asocijativnih kontejnera spadaju `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

Svaki sekvencijalni kontejner ima svoju specifičnu implementaciju.

- `array` je implementiran kao običan statički niz i služi samo da omogući da se klasični statički nizovi koriste na isti način kao i drugi kontejneri.
- `vector` je implementiran preko dinamičkog niza koji se po potrebi realocira.
- `list` je implementiran preko dvostruko povezane liste, dok je `forward_list` implementiran preko jednostruko povezane liste.
- `deque` je specifična struktura podataka implementirana kao vektor u kome se nalaze pokazivači na nizove fiksne veličine.

Adaptori kontejnera samo predstavljaju sloj iznad nekog od postojećih sekvencijalnih kontejnera i pružaju apstraktni interfejs iznad implementacije sekvencijalnog kontejnera, implementirajući funkcije tog interfejsa korišćenjem sekvencijalnog kontejnera za skladištenje podataka. Adaptori imaju svoj podrazumevani sekvencijalni kontejner, koji se može promeniti prilikom deklarisanja promenljivih.

- `stack` implementira funkcije steka korišćenjem vektora za skladištenje podataka. Tip `stack<int>` podrazumeva zapravo `stack<int, vector<int>>`, a moguće je koristiti i, na primer, `stack<int,`

`forward_list<int>>` u kom se za implementaciju steka koristi jednostruko povezana lista.
- `queue` implementira funkcije reda korišćenjem deka za skladištenje podataka. Tip `queue<int>` podrazumeva zapravo `queue<int, deque<int>>`.
- `priority_queue` implementira funkcije reda sa prioritetom korišćenjem vektora za skladištenje podataka. Tip `priority_queue<int>` zapravo predstavlja `priority_queue<int, vector<int>, less<int>>` gde je `less<int>` funkcija koja se koristi za poređenje elemenata i prozrokuje uređenost po opadajućem redosledu prioriteta (na vrhu je element sa najvećim prioritetom). U vektoru su smešteni elementi specijalnog drveta koji se naziva *hip* i koje će biti objašnjeno kasnije.

Asocijativni kontejneri su implementirani pomoću samobalansirajućih uređenih binarnih drveta (obično su to crveno-crna drveta, RBT).

- `set<T>` je implementiran pomoću uređenog binarnog drveta u kome se u čvorovima nalaze elementi skupa i u kom su u svim čvorovima različite vrednosti.
- `multiset<T>` može biti implementiran pomoću uređenog binarnog drveta u kome se u čvorovima nalaze elementi multiskupa i u kom je moguće da postoji više čvorova u kome su različite vrednosti ili pomoću uređenog binarnog drveta u čijim se čvorovima nalaze elementi multiskupa uz njihov broj pojavljivanja, bez ponavljanja elemenata skupa.
- `map<K, V>` je implementiran pomoću uređenog binarnog drveta na osnovu ključeva u kome se u čvorovima nalaze podaci o ključevima i njima pridruženim vrednostima i u kom su u svim čvorovima različite vrednosti ključeva.
- `multimap<K, V>` može biti implementiran pomoću pomoću uređenog binarnog drveta na osnovu ključeva u kome se u čvorovima nalaze podaci o ključevima i njima pridruženim vrednostima i u kom više čvorova može imati istu vrednost ključa.

Neuređeni asocijativni kontejneri su implementirani pomoću heš-tabela.

Dinamički nizovi (vector)

Bez imalo dileme možemo reći da je najkorišćenija struktura podataka niz. 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. Videli smo da se u jeziku C++ za to koristi struktura podataka `vector`. U ovom poglavlju ćemo dati neke naznake kako bi on mogao biti implementiran.

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 (u jeziku C++ ne postoji operator koji bi bio ekvivalentan C funkciji `realloc`).

Implementacija osnovne funkcionalnosti dinamičkog niza u jeziku C++ mogla bi da izgleda ovako (jednostavnosti radi koristimo globalne promenljive i funkcije - one se jednostavno izbegavaju kada se koristi objektno programiranje, ali to nije u fokusu ovog kursa). Ulogu pokazivača `NULL` iz jezika C ima pokazivač `nullptr`, ulogu funkcije `malloc` ima operator `new`, a ulogu funkcije `free` ima operator `delete` (postoje ozbiljne razlike između ovih C funkcija i C++ operatora, ali ni to nije u fokusu ovog kursa).

```
// adresa početka niza
int* a = nullptr;
// broj alociranih i broj popunjenih elemenata
int alocirano = 0, int n = 0;

// čitanje vrednosti elementa na datoj poziciji
// ne proverava se pripadnost opsegu niza
int procitaj(int i) {
  return a[i];
}

// upis vrednosti elementa na datu poziciju
// ne proverava se pripadnost opsegu niza
void postavi(int i, int x) {
  a[i] = x;
}

// pomoćna funkcija koja vrši realokaciju niza na dati
// broj elemenata (veći ili jednak trenutnom broju elemenata)
void realociraj(int m) {
  // alociramo novi niz
  int* novo_a = (int*)new int[m];
  alocirano = m;
  // ako postoji stari niz, kopiramo njegove elemente u novi
  // i brišemo stari niz
  if (a != nullptr) {
    copy_n(a, n, novo_a);
    delete[] a;
  }
  // pokazivač usmeravamo ka novom nizu
  a = novo_a;
}

// dodavanje datog elementa na kraj dinamičkog niza
// niz se automatski realocira ako je potrebno
void dodajNaKraj(int x) {
  if (alocirano <= n)
    realociraj(2 * alocirano + 1);
```

```
  a[n++] = x;
}

// brisanje celog niza
void obrisi() {
  delete[] a;
}
```

Ako se dinamička realokacija vrši na osnovu neke geometrijske strategije za povećanje veličine (npr. svaki put povećamo broj elemenata za 30%), dodavanje na kraj će biti moguće u amortizovanom konstantnom vremenu. Indeksni pristup elementu (pristup na osnovu pozicije) zahteva konstantno vreme (isto kao kod običnog niza).

Liste (list, forward_list)

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
- 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 $O(n)$.

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 (pre svega brisanje sa kraja i iteracija unazad).

Upotreba lista je sve ređa i ređa na savremenim sistemima. Osnovni problem je to što su čvorovi 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 (nizovima i strukturama u kojima se koriste veći blokovi povezanih bajtova). 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.

S obzirom na to da se implementacija lista detaljno pokriva u kursu P2, ona u ovom materijalu neće biti prikazana.

Dekovi (deque)

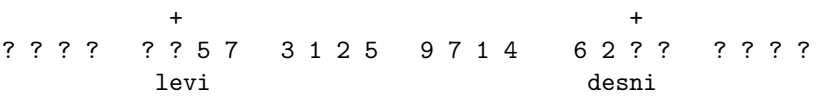
Kao što smo videli u poglavlju o primenama struktura podataka, jedna veoma korisna struktura podataka je red sa dva kraja koja kombinuje funkcionalnost steka i reda.

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 (pod pretpostavkom da održavamo pokazivač 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 (pristup članu na datoj poziciji i) u najgorem slučaju zahteva vreme $O(n)$. Jedini mogući način pretrage je linearna pretraga, čak i kada su elementi u redu sortirani.

U nastavku ćemo prikazati strukturu podataka *dek* (engl. deque) 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 $O(1)$ (doduše amortizovanom). 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. Na primer, red čiji su elementi 5, 7, 3, 1, 2, 5, 9, 7, 1, 4, 6, 2, može biti organizovan u 6 segmenata od po 4 elementa (u praksi je veličina pojedinačnih segmenata obično veća).



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. Posebno

se održavaju pozicije ta dva segmenta. 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 (obično je novi broj segmenata nekoliko puta veći nego polazni, kako bi se naredne realokacije dešavale sve ređe i ređe i kako bi se obezbedilo amortizovano konstantno vreme dodavanja). 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 (oni se prilikom alokacije ne kopiraju, a ne kopiraju se ni sami segmenti). 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.

Prikažimo i jednu moguću implementaciju deka.

```
#include <iostream>

using namespace std;

// jedan segment
struct segment {
  // niz podataka
  int* podaci;
  // broj popunjenih elemenata niza
  int popunjeno;
  // pozicija u nizu na koju se moze ubaciti naredni element
  int tekuci;
};

// alokacija segmenta
```

```
segment* alocirajSegment(int velicina, int smer) {
  // alociramo novi segment
  segment* novi = new segment();
  // alociramo njegove podatke
  novi->podaci = new int[velicina];
  // nijedan podatak još nije upisan
  novi->popunjeno = 0;
  // tekucu poziciju odredjujemo u zavisnosti od smera popunjavanja
  novi->tekuci = smer == 1 ? 0 : velicina - 1;
  return novi;
}

// brisanje segmenta
void obrisiSegment(segment* s) {
  // brisemo podatke u segmentu
  delete[] s->podaci;
  // brisemo segment
  delete s;
}

struct dek {
  // niz pokazivača na segmente
  segment** segmenti;
  // broj segmenata u nizu
  int brojSegmenata;
  // velicina svakog segmenta
  int velicinaSegmenata;
  // pozicije na kojima se nalaze pokazivaci na krajnji levi
  // i kranjnji desni segment (oba mogu biti polupopunjena)
  int levo, desno;
};


// vrši se realociranje deka tako da ima dati broj segmenata
void realocirajDek(dek& d, int brojSegmenata) {
  // pravimo novi niz pokazivača na segmente
  segment** noviSegmenti = new segment*[brojSegmenata];
  // postojeće pokazivače na segmente ćemo smestiti negde oko
  // sredine novog niza
  int uvecanje = (brojSegmenata - d.brojSegmenata) / 2;
  // ako je postojao stari niz pokazivača na segmente
  if (d.segmenti != nullptr)
    // kopiramo te pokazivače u novi niz pokazivača, krenuvši od
    // pozicije uvecanje
    for (int i = 0; i < d.brojSegmenata; i++)
      noviSegmenti[uvecanje + i] = d.segmenti[i];
  // alociramo nove segmente na početku i na kraju novog niza
  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);

  // brišemo stari niz pokazivača na segmente
  delete[] d.segmenti;
  // preusmeravamo pokazivač na niz pokazivača
  d.segmenti = noviSegmenti;
  // ažuriramo broj segmenata
  d.brojSegmenata = brojSegmenata;
  // ažuriramo granice popunjenog dela niza
  d.levo += uvecanje;
  d.desno += uvecanje;
}


// vraća referencu na i-ti element u deku
int& iti(const dek& d, int i) {
  // trenutni broj elemenata u krajnjem levom segmentu
  int uLevom = d.segmenti[d.levo]->popunjeno;
  // pomeramo indeks tako da je pozicija 0 na početku krajnjeg levog
  // segmenta
  i += d.velicinaSegmenata - uLevom;
  // određujemo segment u kom se nalazi traženi element
  segment* s = d.segmenti[d.levo + i / d.velicinaSegmenata];
  // čitamo element iz tog segmenta
  return s->podaci[i % d.velicinaSegmenata];
}

// dodajemo element na početak deka
int dodajNaPocetak(dek& d, int x) {
  // ako je levi segment potpuno popunjen
  if (d.segmenti[d.levo]->popunjeno == d.velicinaSegmenata) {
    // ako levo od njega nema više alociranih segmenata
    if (d.levo == 0)
      // realociramo dek i time alociramo nove segmente
      realocirajDek(d, d.brojSegmenata * 2);
    // prelazimo u prethodni segment
    d.levo--;
  }
  // segment u koji se može upisati element
  segment *s = d.segmenti[d.levo];
  // upisujemo element na tekuću slobodnu poziciju i pomeramo se
  // nalevo
  s->podaci[s->tekuci--] = x;
  // uvećavamo broj popunjenih elemenata
  s->popunjeno++;
}

// dodajemo element na kraj deka
int dodajNaKraj(dek& d, int x) {
  // ako je desni segment potpuno popunjen
```

```
  if (d.segmenti[d.desno]->popunjeno == d.velicinaSegmenata) {
    // ako desno od njega nema više alociranih segmenata
    if (d.desno == d.brojSegmenata - 1)
      // realociramo dek i time alociramo nove segmente
      realocirajDek(d, d.brojSegmenata * 2);
    // prelazimo na naredni segment
    d.desno++;
  }
  // segment u koji se može upisati element
  segment *s = d.segmenti[d.desno];
  // upisujemo element na tekuću slobodnu poziciju i pomeramo se
  // nalevo
  s->podaci[s->tekuci++] = x;
  // uvećavamo broj popunjenih elemenata
  s->popunjeno++;
}

// brisanje deka
void obrisiDek(dek& d) {
  // brisemo sve segmente
  for (int i = 0; i < d.brojSegmenata; i++)
    obrisiSegment(d.segmenti[i]);
  // brisemo niz pokazivaca na segmente
  delete[] d.segmenti;
}

int main() {
  // gradimo prazan dek
  dek d;
  d.velicinaSegmenata = 3;
  d.segmenti = nullptr;
  d.brojSegmenata = 0;
  d.levo = -1;
  d.desno = 0;

  // realociramo ga za 10 elemenata
  realocirajDek(d, 10);

  // dodajemo elemente na pocetak i na kraj
  for (int i = 0; i < 50; i++)
    dodajNaPocetak(d, i);
  for (int i = 50; i < 100; i++)
    dodajNaKraj(d, i);

  // ispisujemo elemente
  for (int i = 0; i < 100; i++)
    cout << iti(d, i) << endl;

  // brisemo dek
  obrisiDek(d);
```

```
  return 0;
}
```

Binarna drveta

Drveta su strukture podataka koje se koriste za predstavljanje hijerarhijskih odnosa između delova (na primer, izraza, sistema direktorijuma i datoteka, organizacije elemenata unutar HTML stranice, sintakse programa unutar prevodioca i slično). U nastavku ćemo se baviti upotrebom drveta za implementaciju struktura podataka čiju smo upotrebu razmotrili u posebnom poglavlju. 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. U programskom jeziku Haskell, to se izražava veoma elegantno: drvo se konstruiše ili pomoću konstruktora (funkcije) `Null` bez parametra ili pomoću konstruktora (funkcije) `Node` čiji su parametri podatak tipa `Int` i dva drveta.

```
{- Tip podataka za predstavljanje uređenog binarnog drveta -}
data Tree =    Null
             | Node Int Tree Tree
```

Tako je `Node 3 (Node 5 Null Null) Null` jedno ispravno formirano drvo tj. podatak tipa `Tree`. Haskell ćemo u ovom materijalu koristi umesto pseudokoda za opis nekih operacija sa drvetima (od čitaoca se ne očekuje poznavanje ovog jezika).

Naravno, podatak ne mora biti tipa `Int` i moguće je definisati i tip podataka kome se prosleđuje tip elemenata unutar drveta (kao što se u C++-u za svaki skup prosleđuje tip podataka tog skupa, kao na primer `set<int>`).

```
{- Tip podataka za predstavljanje uređenog binarnog drveta
   sa podacima tipa a-}
data Tree a =   Null
              | Node a (Tree a) (Tree a)
```

Drvo koje sadrži cele brojeve je onda tipa `Tree Int`.

Uobičajeni način za predstavljanje drveta u jeziku C++ je preko čvorova uvezanih pomoću pokazivača.

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

U slučaju da se drveta koriste za predstavljanje skupova ili multiskupova u čvorovima se čuva samo jedan podatak (element skupa). Ako se koriste za predstavljanje mapa ili multimapa, u čvorovima se čuvaju dva podatka: ključ i vrednost pridružena ključu.

```
struct cvor {
  int k, v;
  cvor *levo, *desno;
}
```

I u jeziku C++ moguće je parametrizovati tip čvora tipom podataka, ali se time nećemo detaljno baviti.

```
template<class K, class V>
struct cvor {
  K k; V v;
  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). Većina funkcija koje ćemo mi implementirati će biti rekurzivna.

Binarna drveta pretrage (uređena binarna drveta)

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.

Naglasimo da nije dovoljno proveriti da je vrednost u svakom čvoru veća od vrednosti u korenu levog poddrveta i vrednosti u korenu desnog poddrveta!

```
bool jeUredjeno(cvor* drvo) {
  if (drvo == nullptr)
     return true;
  if (drvo->levo != null && drvo->x <= drvo->levo->x)
     return false;
  if (drvo->desno != null && drvo->x >= drvo->desno->x)
     return false;
  return jeUredjeno(drvo->levo) && jeUredjeno(drvo->desno);
}
```

Prethodna vraća da je naredno drvo uređeno, što nije tačno, jer je vrednost 4, na pogrešnoj strani čvora 3.

```
      3
  2     5
1   4
```

Ispravna funkcija zahteva izračunavanje najmanje vrednosti desnog poddrveta i najveće vrednosti levog poddrveta.

```
bool jeUredjeno(cvor* drvo) {
  if (drvo == nullptr)
     return true;
  if (drvo->levo != null && drvo->x <= maxVrednost(drvo->levo))
     return false;
  if (drvo->desno != null && drvo->x >= minVrednost(drvo->desno))
     return false;
  return jeUredjeno(drvo->levo) && jeUredjeno(drvo->desno);
}
```

Ako se računanje najmanje i najveće vrednosti uvek izvršava iz početka, posebnim rekurzivnim funkcijama, provera će biti neefikasna. Mnogo bolje rešenje je da se ojača induktivna hipoteza i da funkcija pored provere uređenosti računa minimalnu i maksimalnu vrednost u drvetu.

```
bool jeUredjeno(cvor* drvo, int& minV, int& maxV) {
  if (drvo == nullptr) {
     min = numeric_limits<int>::max();
     max = 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 = minL;
  maxV = maxD;

  return true;
}
```

Umetanje u binarno drvo pretrage

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.

Na primer, ako se duplikati ne dopuštaju, umetanje se može izvršiti narednom funkcijom.

```
-- ubacivanje elementa x u prazno drvo
ubaci x Null = Node x Null Null
-- ubacivanje elementa x u neprazno drvo
ubaci x (Node k l d)
  | x < k   = Node k (ubaci x l) d   -- ubacujemo x levo
  | x > k   = Node k l (ubaci x d)   -- ubacujemo x desno
  | x == k  = Node k l d             -- drvo se ne menja
```

U jeziku C++, implementacija teče po sličnom principu.

```
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;
}
```

Brisanje celog drveta

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

Brisanje iz uređenog binarnog drveta je komplikovanije.

Implementacija u jeziku Haskell je veoma jezgrovita i lako razumljiva.

```
{-
Iz datog drveta se ukljanja čvor sa najmanjom vrednošću.
Funkcija vraća uređen par koji čine obrisana vrednost i izmenjeno drvo.
-}
-- deklaracija funkcije
obrisiMin :: Tree a -> (a, Tree a)
-- ako nema levog poddrveta, briše se element k iz korena, a
-- rezultat je desno poddrvo
obrisiMin (Node k Null d) = (k, d)
-- u suprotnom se briše najmanji element levog poddrveta
-- i dobijeno izmenjeno levo poddrvo se postavlja levo od korena
obrisiMin (Node k l d) = let (x, l') = obrisiMin l
                         in (x, Node k l' d)

{- Iz datog drveta ukljanja dati čvor sa datom vrednošću -}
obrisi :: Ord a => a -> Tree a -> Tree a
-- iz praznog drveta nema šta da se obriše (traženi element ne postoji)
obrisi x Null = Null
-- drvo je neprazno
obrisi x (Node k l d)
 -- ako je element koji se briše manji od korena, briše se iz
 -- levog poddrveta
 | x < k  = Node k (obrisi x l) d
 -- ako je element koji se briše veći od korena, briše se iz
 -- desnog poddrveta
 | x > k  = Node k l (obrisi x d)
 -- ako je element koji se briše jednak korenu, briše se iz korena
 | x == k = case d of
              -- ako desno poddrvo ne postoji, rezultat je levo poddrvo
              Null -> l
              -- u suprotnom se briše najmanji element iz
              -- desnog poddrveta i on se postavlja u koren
              _    -> let (x', d') = obrisiMin d
                      in Node x' l d'
```

Implementacija prethodne funkcionalnosti u jeziku C++ teče po potpuno istom principu, ali je malo komplikovanija.

```
// Iz datog drveta se ukljanja čvor sa najmanjom vrednošću
cvor* obrisiMin(cvor* drvo, int& x) {
   // iz praznog drveta nema šta da se briše
   if (drvo == nullptr) return nullptr;
   // ako je levo poddrvo prazno
   if (drvo->levo == nullptr) {
```

```
      // brišemo koren i vraćamo desno poddrvo
      cvor* desno = drvo->desno;
      x = drvo->x;
      delete drvo;
      return desno;
    }
    // u suprotnom brišemo najmanji element levog poddrveta
    drvo->levo = obrisiMin(drvo->levo, x);
    return drvo;
}

// Iz datog drveta ukljanja dati čvor sa datom vrednošću
cvor* obrisi(cvor* drvo, int x) {
  // iz praznog drveta nema šta da se briše
  if (drvo == nullptr)
    return nullptr;
  // čvor se nalazi levo, pa se tamo i briše
  if (x < drvo->x)
    drvo->levo = obrisi(drvo->levo, x);
  // čvor se nalazi desno, pa se tamo i briše
  else if (x > drvo->x)
    drvo->desno = obrisi(drvo->desno, x);
  // čvor se nalazi u korenu
  else {
    if (drvo->desno == nullptr) {
      // ako je desno poddrvo prazno brišemo koren i vraćamo levo poddrvo
      cvor* levo = drvo->levo;
      delete drvo;
      return levo;
    } else {
      // desno poddrvo nije prazno, pa brišemo najmanji čvor iz
      // njega i vrednost iz tog čvora upisujemo u koren
      int min;
      drvo->desno = obrisiMin(drvo->desno, min);
      drvo->x = min;
    }
  }
  // fizički čvor u kome je koren se nije promenio
  return drvo;
}
```

Balansirana binarna drveta

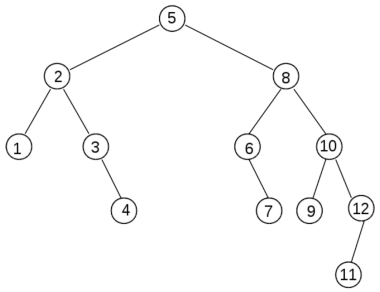
Mana klasičnih uređenih binarnih drveta je to što ako nisu balansirana operacije pretrage, umetanja i brisanja mogu zahtevati linearno vreme u odnosu na broj čvorova u drvetu. Balansirana binarna drveta garantuju da se to ne može dogoditi i da je vremenska složenost najgoreg slučaja ovih operacija logaritamska u odnosu na broj čvorova u drvetu. U nastavku ćemo razmotriti dve najčešće

korišćene vrste balansiranih binarnih drveta.

- Adeljson-Veljski Landisova drveta
- Crveno-crna drveta

AVL

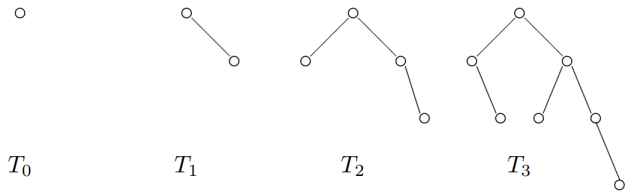
AVL drveta su struktura podataka koja garantuje da složenost ni jedne od operacija traženja, umetanja i brisanja u najgorem slučaju nije veća od $O(\log n)$, gde je n broj elemenata. Posle svake operacije ulaže se dodatni napor da se drvo uravnoteži, tako da visina drveta uvek bude $O(\log n)$. Pri tome 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. Na slici 1 dat je primer jednog AVL drveta.



Slika 1: Primer AVL drveta

Visina h AVL drveta sa n čvorova zadovoljava uslov $h < 2\log_2 n$. Dokažimo ovo tvrđenje indukcijom po visini drveta. Prvo, 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 (jer ako ona ne krše nejednakost, ne krše je ni drveta sa više čvorova zbog monotonosti logaritma). 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*. Nekoliko prvih Fibonačijevih drveta, takvih da je u svakom unutrašnjem čvoru visina levog poddrveta manja, prikazano je na narednoj slici.

Označimo sa n_h minimalni broj čvorova AVL drveta visine h, tj. broj čvorova drveta T_h. Važi $n_0 = 1$, $n_1 = 2$, $n_2 = 4, \ldots$ 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



Slika 2: Fibonačijeva AVL drveta

$h \geq 2$ 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 $O(\log n)$ u odnosu na broj čvorova n.

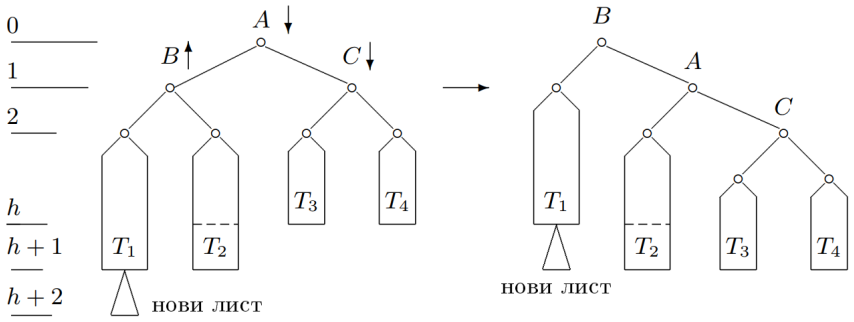
Prilikom umetanja novog elementa u AVL drvo postupa se najpre na način uobičajen za uređeno binarno drvo: pronalazi se mesto čvoru, pa se u drvo dodaje novi list sa ključem jednakim zadatom broju. Čvorovima na putu koji se tom prilikom prelaze odgovaraju razlike visina levog i desnog poddrveta, tzv. *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. Primetimo i da to nije potpuno očigledno, ali se može 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* (ilustrovana na slici 3): koren levog poddrveta B "podiže" se 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 i 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. Čitaocu ostavljamo da se uveri da se rotacijom održavaju svojstva uređenosti drveta.

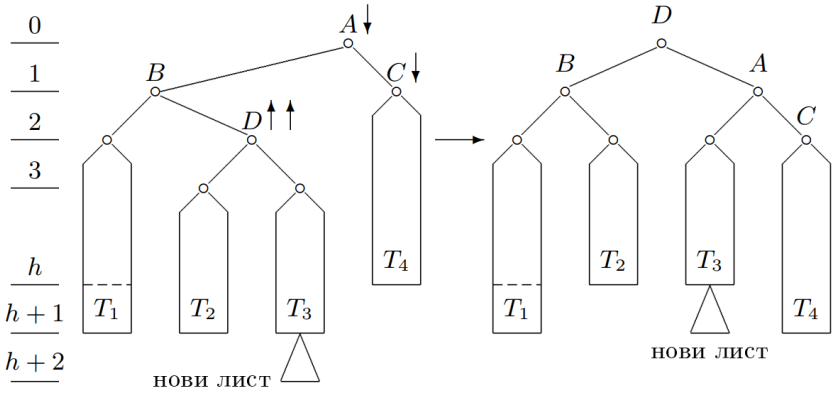
– slučaj kada je novi čvor dodat desnom poddrvetu levog poddrveta je komplikovaniji; tada se drvo može uravnotežiti *dvostrukom rotacijom* (ilustrovanom na slici 4). Novi čvor poddrveta umesto kritičnog čvora postaje desno dete levog deteta kritičnog čvora. Čitaocu ostavljamo da se uveri da se i dvostrukom rotacijom održavaju svojstva uređenosti drveta.



Slika 3: AVL rotacija

Zapazimo da 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. Na primer, da bi se Fibonačijevo drvo F_h sa n čvorova uravnotežilo posle brisanja "loše" izabranog čvora, potrebno je izvršiti $h - 2$, odnosno $O(\log n)$ rotacija. U opštem slučaju je granica za potreban broj rotacija $O(\log n)$. Na sreću, svaka rotacija zahteva konstantni broj koraka, pa je vreme izvršavanja brisanja takođe ograničeno odozgo sa $O(\log n)$.



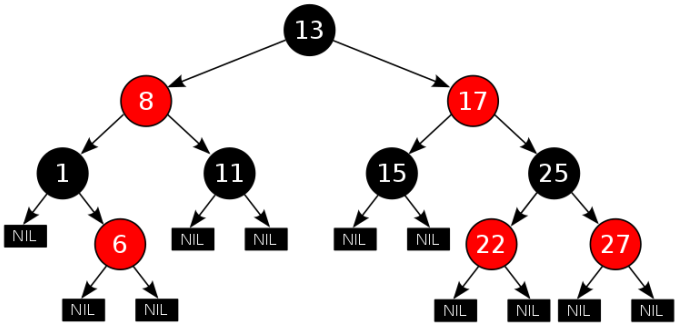
Slika 4: AVL dvostruka rotacija

RBT

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.

Primer jednog takvog drveta je prikazan na slici 5.



Slika 5: Primer RBT

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 (jer na osnovu 4. posle crvenog čvora mora doći crni, a na osnovu 5. sve putanje imaju isti broj crnih čvorova). Ovo svojstvo nam garantuje određeni vid balansiranosti drveta i logaritamsku složenost operacija (pod uslovom da svako umetanje i brisanje u drvo održava nabrojanih 5 invarijanti).

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 kao induktivnu hipotezu 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 (objedini smo unutrašnje čvorove u oba poddrveta i čvor v').

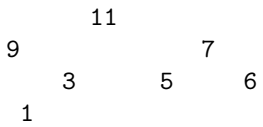
Pošto je bar pola čvorova na svakoj putanji od korena v do listova crno, važi da je $h_b(v)$ bar $h(v)/2$. Zato je na osnovu leme broj unutrašnjih čvorova n veći ili jednak $2^{h(v)/2} - 1$. Zato je $n + 1 \geq 2^{h(v)/2}$, pa je $\log_2(n + 1) \geq h(v)/2$ i važi da je $h(v) \leq 2\log_2(n + 1)$. Zato je visina drveta $O(\log n)$.

Hip

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. Min-hip se definiše analogno, jedino što se zahteva da je vrednost svakog čvora manja ili jednaka od vrednosti u njegovoj deci. Maks-hip omogućava veoma efikasno određivanje maksimalnog elementa u sebi (on se uvek nalazi u korenu), a videćemo i veoma efikasno njegovo uklanjanje. Pošto je i operacija umetanja novog elementa u hip efikasna, ova struktura podataka je veoma dobar kandidat za implementaciju reda sa prioritetom.

Razmotrimo narednu implementaciju maks-hipa. Drvo smeštamo u niz. Koren na poziciju 0, naredna dva elementa na pozicije 1 i 2, zatim naredne elemente

na pozicije 3, 4, 5 i 6 itd. Na primer, hip



predstavljamo nizom

```
11 9 7 4 3 5 6 2 3 1
```

Zahtevamo da se hip popunjava redom i su svi nivoi osim eventualno poslednjeg kompletno popunjeni, a da su u poslednjem nivou popunjeni samo početni elementi.

Lako je uočiti da 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 $\lfloor \frac{i-1}{2} \rfloor$.

```
int roditelj(int i) {
  return (i-1) / 2;
}

int levoDete(int i) {
  return 2*i + 1;
}

int desnoDete(int i) {
  return 2*i + 2;
}
```

Najveći element se uvek nalazi u korenu, tj. na poziciji 0 (što se lako može dokazati indukcijom imajući u vidu da je vrednost u svakom čvoru veća od vrednosti njegovoj u deci).

```
int najveci(const vector<int>& hip) {
  return hip[0];
}
```

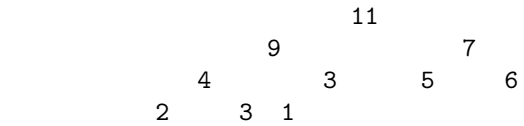
Vreme potrebno za ovo je očigledno $O(1)$.

Razmotrimo kako bismo mogli realizovati operaciju uklanjanja najvećeg elementa iz maks-hipa. Pošto se on nalazi u korenu, a drvo mora biti potpuno popunjeno (osim eventualno poslednjeg nivoa) 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 (i obično nije). Na sreću, popravku možemo izvršiti relativno jednostavno. Potrebno je da uporedimo vrednost u korenu sa vrednošću njegove dece (ako postoje). Ako je vrednost u korenu veća ili jednaka od tih vrednosti, onda koren zadovoljava uslov maks-hipa i procedura može da se završi (jer za sve ostale čvorove znamo

da zadovoljavaju taj uslov, jer je izbacivanje krenulo od ispravnog hipa). U suprotnom, menjamo vrednost u korenu sa većom od vrednosti njegove dece (tj. sa vrednošću njegovog deteta, ako ima samo jedno dete). Nakon toga koren zadovoljava uslov maks-hipa, i preostaje jedino da proverimo (i eventualno popravimo) ono poddrvo u čijem je korenu završila vrednost korena. Ovo je problem istog oblika, samo manje dimenzije u odnosu na polazni i lako se, dakle, rešava induktivno-rekurzivnom konstrukcijom.

```
// element na poziciji k se pomera naniže, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int k) {
  // pozicija najvećeg čvora (razmatrajući roditelja i njegovu decu)
  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;
  // ako roditelj nije najveći
  if (najveci != k) {
    // menjamo roditelja i veće dete
    swap(hip[najveci], hip[k]);
    // rekurzivno obrađujemo veće dete
    pomeriNanize(hip, najveci);
  }
}

// izbacivanje najvećeg elementa iz hipa
int izbaciNajveci(vector<int>& hip) {
  // poslednji element izbacujemo iz hipa i
  // upisujemo ga na početnu poziciju
  hip[0] = hip.back();
  hip.pop_back();
  // pomeramo početni element naniže dok se ne
  // zadovolji uslov hipa
  pomeriNanize(hip, 0);
}
```

Prikažimo rad ovog algoritma na jednom primeru.



Izbacujemo element sa vrha i na njegovo mestu premeštamo poslednji element.

```
2     3
```

Menjamo vrednost u korenu, sa većom od vrednosti njegova dva deteta.

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

Isti postupak primenjujemo na levo poddrvo. Razmenjujemo vrednost u korenu sa većom od vrednosti njegova dva deteta.

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

Postupak se još jednom primenjuje na poddrvo sa korenom `1`.

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

Nakon ovoga, dobijeno drvo predstavlja maks-hip.

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. Dakle, vreme potrebno za uklanjanje najvećeg elementa iz hipa u kojem se nalazi n elemenata je $O(\log n)$.

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. Jedna moguća implementacija je data u nastavku.

```
// element na poziciji k se pomera naviše, razmenom sa svojim
// roditeljem, sve dok se ne zadovolji uslov hipa
void pomeriNavise(vector<int>& hip, int k) {
  // pozicija roditelja čvora k
  int r = roditelj(k);
  // ako čvor k nije koren i ako je veći od roditelja
  if (k > 0 && hip[k] > hip[r]) {
     // razmenjujemo ga sa njegovim roditeljem
     swap(hip[k], hip[r]);
     // pomeramo roditelja navise
     pomeriNavise(hip, r);
```

```
  }
}

// ubacuje se element x u hip
void ubaci(int x) {
  // element dodajemo na kraj
  hip.push_back(x);
  // pomeramo ga naviše dok se ne zadovolji uslov hipa
  pomeriNavise(hip, hip.size() - 1);
}
```

Pošto je i u slučanju operacije pomeranja naniže i u slučaju operacije pomeranja naviše rekurzija repna, ona se može relativno jednostavno ukloniti.

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

void pomeriNanize(vector<int>& hip, int k) {
  while (true) {
    // pozicija najvećeg čvora
    // (razmatrajući roditelja i njegovu decu)
    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;
    // ako je roditelj najveći, uslov hipa je zadovoljen
    if (najveci == k)
      break;
    // menjamo roditelja i veće dete
    swap(hip[najveci], hip[k]);
    // obrađujemo veće dete
    k = najveci;
  }
}
```

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;
  // pretpostavljamo da je levo dete vece
  int veceDete = levoDete(k);
```

```
  // dok god čvor ima dece
  while (veceDete < hip.size()) {
    // poredimo da li je desno dete vece od levog
    int desno = veceDete + 1;
    if (desno < hip.size() && hip[desno] > hip[veceDete])
       veceDete = desno;
    // ako je roditelj veći ili jednak od oba deteta,
    // uslov hipa je zadovoljen
    if (hip[roditelj] >= hip[veceDete])
      break;
    // menjamo roditelja i veće dete
    swap(hip[veceDete], hip[roditelj]);
    // nastavljamo obradu od većeg deteta
    roditelj = veceDete;
    veceDete = levoDete(roditelj);
  }
}
```

Sortiranje uz pomoć hipa (heap sort)

U mnogim situacijama zahteva se formiranje hipa od elemenata nekog unapred zadatog niza. Na primer, algoritam sortiranja pomoću hipa (engl. Heap sort) 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. Interesanto, isti niz (ili vektor) se može koristiti i za smeštanje polaznog niza i za smeštanje hipa i za smeštanje sortiranog niza (elementi hipa se mogu smestiti u početni deo niza, dok se sortirani deo može smestiti u krajnji deo niza). Time se štedi memorijski prostor, međutim, potrebno je malo prilagoditi funkcije za rad sa hipom. Naime, 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 (jer stvarna dimenzija može biti i veća, ako se prostor iza hipa koristi za smeštanje elemenata sortiranog niza). 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 (u istom memorijskom prostoru) je da se krene od praznog hipa i da se jedan po jedan element ubacuje u hip.

```
// formira hip od elemenata niza a dužine n
// hip se smešta u istom memorijskom prostoru kao i niz
void formirajHip(int a[], int n) {
  // ubacujemo jedan po jedan element u hip i pomeramo ga naviše
  for (int i = 1; i < n; i++)
     pomeriNavise(a, i);
}
```

Invarijanta spoljne petlje (tj. induktivna hipoteza) je to da elementi na pozici-

jama $[0, i)$ 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 (elementi na pozicijama $[0, n)$) složeni u maks-hip. Dodatno, razmene ne menjaju multiskup elemenata niza, pa je multiskup elemenata u hipu jednak multiskupu elemenata polaznog niza.

Ovaj način se naziva formiranje hipa naniže ili Vilijamsov metod. Složenost najgoreg slučaja formiranja hipa jednaka je $\Theta(n \log n)$. Naime, složenost operacije umetanja tj. pomeranja elementa naviše u hipu koji ima k elemenata jednaka je $O(\log k)$, pa je ukupna složenost formiranja hipa asimptotski jednaka zbiru svih logaritama od 1 do n, što je, kako smo ranije videli $\Theta(n \log n)$.

Drugi, efikasniji način formiranja hipa naziva se formiranje hipa naviše ili 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. Na primer, ako je dat niz

```
7 2   8 5 4 6 5 1 3
```

svi elementi osim prva dva čine korenove ispravnih hipova. Zaista, u nizu je smešteno sledeće drvo.

```
          7
      2       8
   5    4  6   5
 1   3
```

Elementi od 4 do 3 nemaju naslednike i trivijalno predstavljaju ispravne jednočlane hipove. I u opštem slučaju, svi elementi u drugoj polovini niza predstavljaju listove i za njih je ova invarijanta trivijalno ispunjena. Element 5 je veći od oba svoja sina, pa je koren ispravnog maks-hipa. Slično, element 8 je veći od oba svoja sina, pa je i on koren ispravnog maks-hipa.

Postavlja se pitanje 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.

```
// formira hip od elemenata niza a dužine n
// hip se smešta u istom memorijskom prostoru kao i niz
void formirajHip(int a[], int n) {
   // sve elemente osim listova pomeramo naniže
   for (int i = n/2; i >= 0; i--)
      pomeriNanize(a, n, i);
}
```

Ocenimo složenost ovog algoritma. Prvo, veoma je zgodno to što je jedna

polovina elemenata niza već na svom mestu. Spuštanje elemenata niz hip ima složenost $O(\log n)$, pa je ovde ponovo u pitanju neki zbir logaritamskih složenosti i na prvi pogled može se pomisliti da će i ovde složenost biti $\Theta(n \log n)$ - ona će svakako biti $O(n \log n)$, ali ćemo pokazati da će biti niža tj. $\Theta(n)$. Naime, 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 i tako dalje. Ukupan broj koraka za hip visine i bi bio $i + 2(i-1) + 4(i-2) + ... 2^{i-1} \cdot 1$ tj.

$$\sum_{k=0}^{i-1} 2^k (i-k).$$

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 i $H(0) = 0$. Razmotajmo ovu rekurentnu jednačinu.

$$\begin{aligned}
H(i) &= 2H(i-1) + i \\
&= 2(2H(i-2) + (i-1)) + i = 4H(i-2) + 2(i-1) + i \\
&= 4(2H(i-3) + (i-2)) + 2(i-1) + i = 8H(i-3) + 4(i-2) + 2(i-1) + i \\
&= ... \\
&= 2^i H(0) + \sum_{k=0}^{i-1} 2^k (i-k) = \sum_{k=0}^{i-1} 2^k (i-k) \\
&= i \sum_{k=0}^{i-1} 2^k - \sum_{k=0}^{i-1} k 2^k = i(2^i - 1) - ((i-1)-1)2^{(i-1)+1} - 2 = 2^{i+1} - i - 2
\end{aligned}$$

Poslednje važi jer znamo da je

$$\sum_{k=0}^{n} 2^k = 2^{n+1} - 1, \quad \sum_{k=0}^{n} k 2^k = (n-1)2^{n+1} + 2.$$

Prvo sledi na osnovu formule za zbir geometrijskog reda $\sum_{k=0}^{n} x^k$, a drugo njenim diferenciranjem i množenjem sa x.

Postoji i lakši način da se dođe do istog rezultata. Naime, jednačinu $H(i) = 2H(i-1)+i$ možemo zapisati kao $H(i)+i = 2(H(i-1)+(i-1))+2$ i onda uvesti

smenu $G(i) = H(i)+i$. Zato je $G(i) = 2G(i-1)+2$ i $G(0) = 0$. Odmotavanjem sada dobijamo

$$\begin{aligned} G(i) &= 2G(i-1)+2 \\ &= 2(2G(i-2)+2)+2 = 4G(i-2)+2\cdot 2+2 \\ &= 4(2G(i-3)+2)+2\cdot 2+2 = 8G(i-3)+4\cdot 2+2\cdot 2+2 \\ &= ... \\ &= 2^i G(0) + 2\sum_{k=0}^{i-1} 2^k = 2^{i+1}-2 \end{aligned}$$

Zato je $H(i) = G(i) - i = 2^{i+1} - i - 2$.

Pošto je $G(i) + 2 = 2(G(i-1)+2)$, smenom $F(i) = G(i)+2$ bi se dobilo da je $F(i) = 2F(i-1)$ i $F(0) = 2$, pa bi skoro direktno sledilo da je $F(i) = 2^{i+1}$ i $G(i) = F(i) - 2 = 2^{i+1} - 2$.

Od svih ovih 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!

Ispitajmo koliko je ta razlika u praksi značajna. Za niz od 10^9 nasumično odabarnih brojeva formiranje naviše traje oko $1,902$ sekundi, a formiranje naniže oko $1,943$ sekundi. Razlika je praktično beznačajna, naročito u svetlu toga što kasnije vađenje elemenata iz hipa koje takođe ima asimptotsku složenost $\Theta(n \log n)$ traje dodatnih oko 50 sekundi i apsolutno dominira sortiranjem. Stvar je malo drugačija kada se krene od već sortiranog niza, jer tada formiranje naviše traje oko $0,782$ sekunde, a naniže oko $2,534$ sekunde, međutim, opet kasnija faza dominira i traje oko 11 dodatnih sekundi, pa ušteda ni u ovom slučaju nije naročito značajna. Objašnjenje možemo naći u tome što se kod operacije pomeranja naniže vrši više poređenja nego kod operacije pomeranja elemenata naviše, tako da iako se operacija pomeranja naviše malo češće vrši, ona je sama po sebi brža. Takođe, razlika u asimptotskim klasama n i $n \log n$ je prilično mala i nije je uvek lako uhvatiti u praksi, jer konstante koje zanemarujemo mogu značajno uticati na vreme izvršavanja.

Prikažimo na kraju i implementaciju 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 (elementi u intervalu $[0, n-1]$ čine ispravan hip, dok je interval $(n-1, n)$ prazan). 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 (hip je za jedan element kraći, a sortirani deo niza je za jedan element duži). 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.

```
// sortira se niz a dužine n
void hipSort(int a[], int n) {
  // formiramo hip na osnovu elemenata niza
  formirajHip(a, n);
  for (int i = n-1; i > 0; i--) {
    // najveći element vadimo iz hipa i ubacujemo ga na početak
    // sortiranog dela niza
    swap(a[0], a[i]);
    // pomeramo koren hipa naniže, popravljajući hip
    // nakon izbacivanja najvećeg, broj elemenata hipa jednak je i
    pomeriNanize(a, i, 0);
  }
}
```

Heširanje

Pogledati opis u knjizi profesora Živkovića. U jeziku C++ heširanje se koristi za implementaciju tzv. neuređenih kolekcija (`unordered_set` i `unordered_map`). Na raspolaganju nam je i struktura `hash` koja je specijalizovana za sve osnovne tipove podataka i koja se može upotrebiti za računanje heš-vrednosti. Na primer,

```
hash<string> h;
cout << h("algoritmi") << endl;
```

ispisuje 3697056665034372007.

Čas 6.1, 6.2, 6.3 - još neke korisne struktre podataka

Prefiksno drvo

Uređena binarna drveta omogućavaju efikasnu implementaciju struktura sa asocijativnim pristupom kod kojih se pristup vrši po ključu koji nije celobrojna vrednost, već string ili nešto drugo. Još jedna struktura u vidu drveta koja omogućava efikasan asocijativni pristup je *prefiksno drvo* takođe poznato pod

engleskim nazivom *trie* (od engleske reči *reTRIEeval*). Osnovna ideja ove strukture je da putanje od korena do listova ili do nekih od unutrašnjih čvorova, kodiraju ključeve, a da se podaci vezani za taj ključ čuvaju u čvoru do kojeg se dolazi pronalaženjem ključa duž putanje. U slučaju niski, koren sadrži praznu reč, a prelaskom preko svake grane se na do tada formiranu reč nadovezuje još jedan karakter. Pritom, zajednički prefiksi različitih ključeva su predstavljeni istim putanjama od korena do tačke razlikovanja. Jedan primer ovakvog drveta, kod kojeg su prikazane oznake pridružene granama, a ne čvorovima, je sledeći:

```
    a     n    d
  n   t   o   a   u
 a       ć ž  n  h ž
```

Ključevi koje drvo čuva su *ana*, *at*, *noć*, *nož*, *da*, *dan*, *duh* i *duž*. Primetimo da se ključ *da* ne završava listom i da stoga svaki čvor mora čuvati informaciju o tome da li se njime kompletira neki ključ (i u tom slučaju sadržati podatak) ili ne. Ilustracije radi, mogu se prikazati oznake na čvorovima, koje predstavljaju prefikse akumulirane do tih čvorova. Treba imati u vidu da ovaj prikaz ne ilustuje implementaciju, već samo prefikse duž grane. Simbol @ predstavlja praznu reč.

```
             @
   a         n          d
 an   at     no     da     du
ana        noć  nož  dan  duh  duž
```

U slučaju da neki čvor ima samo jednog potomka i ne predstavlja kraj nekog ključa, grana do njega i grana od njega se mogu spojiti u jednu, njihovi karakteri nadovezati, a čvor eliminisati. Ovako se dobija kompaktnija reprezentacija prefiksnog drveta.

Pored opšteg asocijativnog pristupa podacima, očigledna primena ove strukture je i implementacija konačnih rečnika, na primer u svrhe automatskog kompletiranja ili provere ispravnosti reči koje korisnik kuca na računaru ili mobilnom telefonu.

Napomenimo još i da ova struktura nije rezervisana za čuvanje stringova. Na primer, u slučaju celih brojeva ili brojeva u pokretnom zarezu, ključ mogu biti niske bitova koje predstavljaju takve brojeve.

U slučaju konačne azbuke veličine m, složenost operacija u najgorem slučaju je $O(mn)$, gde je n dužina reči koja se traži, umeće ili briše. Pretraga i umetanje se pravolinijski implementiraju, dok je u slučaju brisanja nekada potrebno brisati više od jednog čvora.

Ukoliko su ključevi relativno kratki, prednost prefiksnog drveta je što složenost zavisi od dužine zapisa ključa, a ne od broja elemenata u drvetu. Mana je potreba za čuvanjem pokazivača uz svaki karakter u drvetu.

U nastavku su date implementacije osnovnih operacija sa prefiksnim drvetom

na primeru formiranja i pretrage rečnika.

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
using namespace std;

// struktura čvora prefiksnog drveta - u svakom čvoru čuvamo niz
// grana obeleženih karakterima ka potomcima i informaciju da li
// je u ovom čvoru kraj neke reči

struct cvor {
  bool krajKljuca = false;
  unordered_map<char, node*> grane;
};


// tražimo sufiks reči w koji počinje od pozicije i u drvetu na
// čiji koren ukazuje pokazivač trie
bool nadji(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan, on je u korenu akko je u korenu obeleženo
  // da je tu kraj reči
  if (i == w.size())
    return drvo->krajKljuca;

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako je nađemo, rekurzivno tražimo ostatak sufiksa od pozicije i+1
  if(it!=drvo->grane.end())
    return find(it->second, w, i+1);

  // nismo našli granu sa w[i], pa reč ne postoji
  return false;
}

// tražimo u drvetu na čiji koren ukazuje pokazivač trie reč w
bool nadji(cvor *drvo, const string& w) {
  return nadji(drvo, w, 0);
}

// umeće sufiks reči w od pozicije i u drvo na čiji koren ukazuje
// pokazivač trie
void umetni(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan samo u korenu beležimo da je tu kraj reči
  if (i == w.size()) {
    drvo->krajKljuca = true;
    return;
  }
```

```
  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako takva grana ne postoji, dodajemo je kreirajući novi čvor
  if(it==drvo->grane.end())
    drvo->grane[w[i]]=new cvor();

  // sada znamo da grana sa w[i] sigurno postoji i preko te grane
  // nastavljamo dodavanje sufiksa koji počinje na i+1;
  umetni(drvo->grane[w[i]].second, w, i+1);
}

// umeće reč w u drvo na čiji koren ukazuje pokazivač w
void umetni(cvor *drvo, string& w) {
  return umetni(drvo, w, 0);
}

// program kojim testiramo gornje funkcije
int main() {
  cvor* drvo = new cvor();
  vector<string> reci
    {"ana", "at", "noc", "noz", "da", "dan", "duh", "duz"};
  vector<string> reci_neg
    {"", "a", "d", "ananas", "marko", "ptica"};
  for(auto w : reci)
    umetni(drvo, w);
  for(auto w : reci)
    cout << w << ": " << (nadji(drvo, w) ? "da" : "ne") << endl;
  for(auto w : reci_neg)
    cout << w << ": " << (nadji(drvo, w) ? "da" : "ne") << endl;
  return 0;
}
```

Disjunktni podskupovi (union-find)

Ponekada je potrebno održavati u programu nekoliko disjunktnih podskupova određenog skupa, pri čemu je potrebno moći za dati element efikasno pronaći kom skupu pripada (tu operaciju zovemo `find`) i efikasno spojiti dva zadata podskupa u novi, veći podskup (tu operaciju zovemo `union`). Pomoću operacije `find` lako možemo za dva elementa proveriti da li pripadaju istom podskupu tako što za svaki od njih pronađemo oznaku podskupa i proverimo da li su one jednake.

Jedna moguća implementacija je da se održava preslikavanje svakog elementa u oznaku podskupa kojem pripada. Ako pretpostavimo da su svi elementi numerisani brojevima od 0 do $n-1$, onda ovo preslikavanje možemo realizovati pomoću običnog niza gde se na poziciji svakog elementa nalazi oznaka podskupa kojem on pripada. Operacija `find` je tada trivijalna (samo se iz niza pročita oznaka podskupa) i složenost joj je $O(1)$. Operacija `union` je mnogo sporija jer

zahteva da se oznake svih elemenata jednog podskupa promene u oznake drugog, što zahteva da se prođe kroz ceo niz i složenosti je $O(n)$.

```
int id[MAX_N];
int n;

void inicijalizuj() {
  for (int i = 0; i < n; i++)
    id[i] = i;
}

int predstavnik(int x) {
   return id[x];
}

int ekvalentni(int x, int y) {
  return predstavnik(x) == predstavnik(y);
}

int unija(int x, int y) {
  int idx = id[x], idy = id[y];
  for (int i = 0; i < n; i++)
    if (id[i] == idx)
      id[i] = idy;
}
```

Ključna ideja je da elemente ne preslikavamo u oznake podskupova, već da podskupove čuvamo u obliku drveta tako da svaki element slikamo u njegovog roditelja u drvetu. Korene drveta ćemo slikati same u sebe i smatrati ih oznakama podskupova. Dakle, da bismo na osnovu proizvoljnog elementa saznali oznaku podskupa kom pripada, potrebno je da prođemo kroz niz roditelja sve dok ne stignemo do korena. Naglasimo da su u ovim drvetima pokazivači usmereni od dece ka roditeljima, za razliku od klasičnih drveta gde pokazivači ukazuju od roditelja ka deci.

Prvi algoritam odgovara situaciji u kojoj osoba koja promeni adresu obaveštava sve druge osobe o svojoj novoj adresi. Drugi odgovara situaciji u kojoj samo na staroj adresi ostavlja informaciju o svojoj novoj adresi. Ovo, naravno, malo usporava dostavu pošte, jer se mora preći kroz niz preusmeravanja, ali ako taj niz nije predugačak, može biti značajno efikasnije od prvog pristupa.

Uniju možemo vršiti tako što koren jednog podskupa usmerimo ka korenu drugog.

```
int roditelj[MAX_N];
int n;

void inicijalizuj() {
  for (int i = 0; i < n; i++)
    roditelj[i] = i;
```

```
}

int predstavnik(int x) {
  while (roditelj[x] != x)
    x = roditelj[x];
  return x;
}

int unija(int x, int y) {
  int fx = predstavnik(x), fy = predstavnik(y);
  roditelj[fx] = fy;
}
```

Složenost prethodnog pristupa zavisi od toga koliko su drveta kojima se predstavljaju podskupovi balansirana. U najgorem slučaju se ona mogu izdegenerisati u listu i tada je složenost svake od operacija $O(n)$. Ilustrujmo ovo jednim primerom.

```
0  1  2  3  4  5  6  7

unija 7 6

0  1  2  3  4  5  6  6

unija 6 5

0  1  2  3  4  5  5  6

unija 5 4

0  1  2  3  4  4  5  6

unija 4 3

0  1  2  3  3  4  5  6

unija 3 2

0  1  2  2  3  4  5  6

unija 2 1

0  1  1  2  3  4  5  6

unija 1 0

0  0  1  2  3  4  5  6
```

Upit kojim se traži prednstavnik skupa kojem pripada element 7 se realizuje nizom koraka kojima se prelazi preko sledećih elemenata 7, 6, 5, 4, 3, 2, 1, 0. Iako ovo deluje lošije od prethodnog pristupa, gde je bar pronalaženje podskupa koštalo $O(1)$, kada su drveta izbalansirana, tada je složenost svake od operacija $O(\log n)$ i centralni zadatak da bi se na ovoj ideji izgradila efikasna struktura podataka je da se nekako obezbedi da drveta ostanu izbalansirana. Ključna ideja je da se prilikom izmena (a one se vrše samo u sklopu operacije unije), ako je moguće, obezbedi da se visina drveta kojim je predstavljena unija ne poveća u odnosu na visine pojedinačnih drveta koja predstavljaju skupove koji se uniraju (visinu možemo definisati kao broj grana na putanji od tog čvora do njemu najudaljenijeg lista). Prilikom pravljenja unije imamo slobodu izbora korena kog ćemo usmeriti prema drugom korenu. Ako se uvek izabere da koren plićeg drveta usmeravamo ka dubljem, tada će se visina unije povećati samo ako su oba drveta koja uniramo iste visine. Visinu drveta možemo održavati u posebnom nizu koji ćemo iz razloga koji će biti kasnije objašnjeni nazvati `rang`.

```
int roditelj[MAX_N]; int n;
int rang[MAX_N];

void inicijalizuj() {
  for (int i = 0; i < n; i++) {
    roditelj[i] = i;
    rang[i] = 0;
  }
}

int predstavnik(int x) {
  while (roditelj[x] != x)
    x = roditelj[x];
  return x;
}

int unija(int x, int y) {
  int fx = predstavnik(x), fy = predstavnik(y);
  if (rang[fx] < rang[fy])
     roditelj[fx] = fy;
  else if (rang[fy] < rang[fx])
     roditelj[fy] = fx;
  else {
     roditelj[fx] = fy;
     rang[fx]++;
  }
}
```

Prikažimo rad algoritma na jednom primeru. Podskupove ćemo predstavljati drvetima.

1 2 3 4 5 6 7 8

```
unija 1 2

1  3  4  5  6  7  8
2

unija 6 7

1  3  4  5  6  8
2          7

unija 4 7

1  3  5     6    8
2        4   7

unija 5 8

1  3    6     8
2    4   7   5

unija 1 3

  1       6     8
2   3   4   7   5

unija 5 4

  1        6
2   3   4  7  8
             5

unija 3 7

       6
  1  4  7  8
2   3     5
```

Dokažimo indukcijom da se u drvetu čiji je koren na visini h nalazi bar 2^h čvorova. Baza je početni slučaj u kome je svaki čvor svoj predstavnik. Visina svih čvorova je tada nula i sva drveta imaju $2^0 = 1$ čvor. Pokažimo da svaka unija održava ovu invarijantu. Po induktivnoj hipotezi pretpostavljamo da oba drveta koja predstavljaju podskupove koji se uniraju imaju visine h_1 i h_2 i bar 2^{h_1} i 2^{h_2} čvorova. Ukoliko se uniranjem visina ne poveća, invarijanta je očuvana jer se broj čvorova uvećao. Jedini slučaj kada se povećava visina unije je kada je $h_1 = h_2$ i tada unirano drvo ima visinu $h = h_1 + 1 = h_2 + 1$ i bar $2^{h_1} + 2^{h_2} = 2^h$ čvorova. Time je tvrđenje dokazano. Dakle, složenost

svake operacije pronalaženja predstavnika u skupu od n čvorova je $O(\log n)$, a pošto uniranje nakon pronalaženja predstavnika vrši još samo $O(1)$ operacija, i složenost nalaženja unije je $O(\log n)$.

Recimo i da je umesto visine moguće održavati i broj čvorova u svakom od podskupova. Ako uvek usmeravamo predstavnika manjeg ka predstavniku većeg podskupa, ponovo ćemo dobiti logaritamsku složenost najgoreg slučaja za obe operacije.

Iako je ova složenost sasvim prihvatljiva (složenost nalaženja n unija je $O(n \log n)$), može se dodatno poboljšati veoma jednostavnom tehnikom poznatom kao *kompresija putanje*. Naime, prilikom pronalaženja predstavnika možemo sve čvorove kroz koje prolazimo usmeriti ka korenu. Jedan način da se to uradi je da se nakon pronalaženja korena, ponovo prođe kroz niz roditelja i svi pokazivači usmere ka korenu.

```
int predstavnik(int x) {
  int koren = x;
  while (koren != roditelj[koren])
    koren = roditelj[koren];
  while (x != koren) {
    int tmp = roditelj[x];
    roditelj[x] = koren;
    x = tmp;
  }
  return koren;
}
```

Za sve čvorove koji se obilaze od polaznog čvora do korena, dužine putanja do korena se nakon ovoga smanjuju na 1, međutim, kao što primećujemo, niz rangova se ne menja. Ako rangove tumačimo kao broj čvorova u drvetu, onda se prilikom kompresije staze ta statistika i ne menja, pa je postupak korektan. Ako rangove tumačimo kao visine, jasno je da prilikom kompresije putanje niz visina postaje neažuran. Međutim, interesantno je da ni u ovom slučaju nema potrebe da se on ažurira. Naime, brojevi koji se sada čuvaju u tom nizu se više ne smatraju visinama čvorova, već prosto njihovim rangovima tj. pomoćnim podacima koji nam pomažu da preusmerimo čvorove prilikom uniranja. Pokazuje se da se ovim ne narušava složenost najgoreg slučaja i da funkcija nastavlja korektno da radi.

U prethodnoj implementaciji se dva puta prolazi kroz putanju od čvora do korena. Ipak, slične performanse se mogu dobiti i samo u jednom prolazu ako se svaki čvor usmeri ka roditelju svog roditelja. Za sve čvorove koji se obilaze od polaznog čvora do korena, dužine putanja do korena se nakon ovoga smanjuju dvostruko, što je dovoljno za odlične performanse.

```
int predstavnik(int x) {
  while (x != roditelj[x]) {
    x = roditelj[x]
```

```
    roditelj[x] = roditelj[roditelj[x]]
  }
  return x;
}
```

Primetimo da je ovim dodata samo jedna linija koda u prvobitnu implementaciju. Ovom jednostavnom promenom agregatna složenost operacija postaje samo $O(\alpha(n))$, gde je $\alpha(n)$ inverzna Akermanova funkcija koja strašno sporo raste. Za bilo koji broj n koji je manji od broja atoma u celom univerzumu važi da je $\alpha(n) < 5$, tako da je vreme praktično konstantno.

Problem: Logička matrica u početku sadrži sve nule. Nakon toga se nasumično dodaje jedna po jedna jedinica. Kretanje po matrici je moguće samo po jedinicama i to samo na dole, na gore, na desno i na levo. Napisati program koji učitava dimenziju matrice, a zatim poziciju jedne po jedne jedinice i određuje nakon koliko njih je prvi put moguće sići od vrha do dna.

Osnovna ideja je da se formiraju svi podskupovi elemenata između kojih postoji put. Kada se uspostavi veza između dva elementa takva dva podskupa, podskupovi se spajaju. Provera da li postoji put između dva elementa svodi se onda na proveru da li oni pripadaju istom podskupu. Podskupove možemo čuvati na način koji smo opisali. Putanja od vrha do dna postoji ako postoji putanja od bilo kog elementa u prvom redu matrice do bilo kog elementa u dnu matrice. To bi dovelo do toga da u svakom koraku moramo da proveravamo sve parove elemenata iz gornjeg i donjeg reda. Međutim, možemo i bolje. Dodaćemo veštački početni čvor (nazovimo ga izvor) i spojićemo ga sa svim čvorovima u prvoj vrsti matrice i završni čvor (nazovimo ga ušće) i spojićemo ga sa svim čvorovima u poslednjoj vrsti matrice. Tada se u svakom koraku samo može proveriti da li su izvor i ušće spojeni.

```
// redni broj elementa (x, y) u matrici
int kod(int x, int y, int n) {
  return x*n + y;
}

int put(int n, const vector<pair<int, int>>& jedinice) {
  // alociramo matricu n*n
  vector<vector<int>> a(n);
  for (int i = 0; i < n; i++)
    a[i].resize(n);

  // dva dodatna veštačka čvora
  const int izvor = n*n;
  const int usce = n*n+1;

  // inicijalizujemo union-find strukturu za sve elemente matrice
  // (njih n*n), izvor i ušće
  inicijalizacija(n*n + 2);
```

```
  // spajamo izvor sa svim elementima u prvoj vrsti matrice
  for (int i = 0; i < n; i++)
    unija(izvor, kod(0, i, n));

  // spajamo sve elemente u poslednjoj vrsti matrice sa ušćem
  for (int i = 0; i < n; i++)
    unija(kod(n-1, i), usce);

  // broj obrađenih jedinica
  int k = 0;
  while (k < jedinice.size()) {
    // čitamo narednu jedinicu
    int x = jedinice[k].first, y = jedinice[k].second;
    k++;
    // ako je u matrici već jedinica, nema šta da se radi
    if (a[x][y] == 1) continue;
    // upisujemo jedinicu u matricu
    a[x][y] = 1;
    // povezujemo podskupove u sva četiri smera
    if (x > 0 && a[x-1][y])
      unija(kod(x, y, n), kod(x-1, y, n));
    if (x + 1 < n && a[x+1][y])
      unija(kod(x, y, n), kod(x+1, y, n));
    if (y > 0 && a[x][y-1])
      unija(kod(x, y, n), kod(x, y-1, n));
    if (y + 1 > 0 && a[x][y+1])
      unija(kod(x, y, n), kod(x, y+1, n));
    // proveravamo da li su izvor i ušće spojeni
    if (predstavnik(izvor) == predstavnik(usce))
      return k;
  }

  // izvor i ušće nije moguće spojiti na osnovu datih jedinica
  return 0;
}
```

Upiti raspona

Određene strukture podataka su posebno pogodne za probleme u kojima se traži da se nad nizom elemenata izvršavaju upiti koji zahtevaju izračunavanje statistika nekih raspona tj. segmenata niza (engl. range queries).

Problem: Implementirati strukturu podataka koja obezbeđuje efikasno izračunavanje zbirova segmenata datog niza određenih intervalima pozicija $[a, b]$.

Rešenje grubom silom koje bi smestilo sve elemente u klasičan niz i pri svakom upitu iznova računalo zbir elemenata na pozicijama iz datog intervala imalo bi složenost $O(mn)$, gde je m broj upita, a n dužina niza, što je u slučaju dugačkih

nizova i velikog broja upita nedopustivo neefikasno. Jednostavno rešenje je zasnovano na ideji koju smo već ranije razmatrali. Umesto čuvanja elemenata niza, možemo čuvati niz zbirova prefiksa niza. Zbir svakog segmenta $[a, b]$ možemo razložiti na razliku prefiksa do elementa b i prefiksa do elementa $a - 1$. Svi prefiksi se mogu izračunati u vremenu $O(n)$ i smestiti u dodatni (a ako je ušteda memorije bitna, onda čak i u originalni) niz. Nakon ovakvog pretprocesiranja, zbir svakog segmenta se može izračunati u vremenu $O(1)$, pa je ukupna složenost $O(n + m)$.

Donekle srodan problem može biti i sledeći.

Problem: Dat je niz dužine n koji sadrži samo nule. Nakon toga izvršavaju se upiti oblika $([a, b], x)$, koji podrazumevaju da se svi elementi na pozicijama iz intervala $[a, b]$ uvećaju za vrednost x. Potrebno je odgovoriti kako izgleda niz nakon izvršavanja svih tih upita.

Rešenje grubom silom bi u svakom koraku u petlji uvećavalo sve elemente na pozicijama $[a, b]$. Složenost tog naivnog pristupa bila bi $O(mn)$, gde je m broj upita, a n dužina niza.

Mnogo bolje rešenje se može dobiti ako se umesto elemenata niza pamte razlike između svaka dva susedna elementa niza. Ključni uvid je da se tokom uvećavanja svih elemenata niza menjaju samo razlike između elemenata na pozicijama a i $a - 1$ (ta razlika se uvećava za x) kao i između elemenata na pozicijama $b + 1$ i b (ta razlika se umanjuje za x). Ako znamo sve elemente niza, tada niz razlika susednih elemenata možemo veoma jednostavno izračunati u vremenu $O(n)$. Sa druge strane, ako znamo niz razlika, tada originalni niz možemo takođe veoma jednostavno rekonstruisati u vremenu $O(n)$. Jednostavnosti radi, možemo pretpostaviti da početni niz proširujemo sa po jednom nulom sa leve i desne strane.

Može se primetiti da se rekonstrukcija niza vrši zapravo izračunavanjem prefiksnih zbirova niza razlika susednih elemenata, što ukazuje na duboku vezu između ove dve tehnike. Zapravo, razlike susednih elemenata predstavljaju određeni diskretni analogon izvoda funkcije, dok prefiksni zbirovi onda predstavljaju analogiju određenog integrala. Izračunavanje zbira segmenta kao razlike dva zbira prefiksa odgovara Njutn-Lajbnicovoj formuli.

Dakle, niz zbirova prefiksa, omogućava efikasno postavljanje upita nad segmentima niza, ali ne omogućava efikasno ažuriranje elemenata niza, jer je potrebno ažurirati sve zbirove prefiksa nakon ažuriranog elementa, što je naročito neefikasno kada se ažuriraju elementi blizu početka niza (složenost najgoreg slučaja $O(n)$). Niz razlika susednih elemenata dopušta stalna ažuriranja niza, međutim, izvršavanje upita očitavanja stanja niza podrazumeva rekonstrukciju niza, što je složenosti $O(n)$.

Problemi koje ćemo razmatrati u ovom poglavlju su specifični po tome što omogućavaju da se upiti ažuriranja niza i očitavanja njegovih statistika javljaju isprepletano. Za razliku od prethodnih, statičkih upita nad rasponima

(engl. static range queries), ovde ćemo razmatrati tzv. dinamičke upite nad rasponima (engl. dynamic range queries), tako da je potrebno razviti naprednije strukture podataka koje omogućavaju izvršavanje oba tipa upita efikasno. Na primer, razmotrimo sledeći problem.

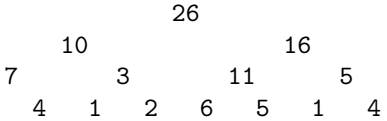
Problem: Implementirati strukturu podataka koja obezbeđuje efikasno izračunavanje zbirova segmenata datog niza određenih intervalima pozicija $[a, b]$, pri čemu se pojedinačni elementi niza često mogu menjati.

U nastavku ćemo videti dve različite, ali donekle slične strukture podataka koje daju efikasno rešenje prethodnog problema i njemu sličnih.

Segmentna drveta

Jedna struktura podataka koja omogućava prilično jednostavno i efikasno rešavanje ovog problema su *segmentna drveta*. Opet se tokom faze pretprocesiranja izračunavaju zbirovi određenih segmenata polaznog niza, a onda se zbir elemenata proizvoljnog segmenta polaznog niza izražava u funkciji tih unapred izračunatih zbirova. Recimo i da segmentna drveta nisu specifična samo za sabiranje, već se mogu koristiti i za druge statistike segmenata koje se izračunavaju asocijativnim operacijama (na primer za određivanje najmanjeg ili najvećeg elementa, nzd-a svih elemenata i slično).

Pretpostavimo da je dužina niza stepen broja 2 (ako nije, niz se može dopuniti do najbližeg stepena broja 2, najčešće nulama). Članovi niza predstavljaju listove drveta. Grupišemo dva po dva susedna čvora i na svakom narednom nivou drveta čuvamo roditeljske čvorove koji čuvaju zbirove svoja dva deteta. Ako je dat niz $3, 4, 1, 2, 6, 5, 1, 4$, segmentno drvo za zbirove izgleda ovako.



Pošto je drvo potpuno, najjednostavnija implementacija je da se čuva implicitno u nizu (slično kao u slučaju hipa). Pretpostavićemo da elemente drveta smeštamo od pozicije 1, jer je tada aritmetika sa indeksima malo jednostavnija (elementi polaznog niza mogu biti indeksirani klasično, krenuvši od nule).

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
-  26 10 16 7  3  11 5  3  4  1  2  6  5  1  4
```

Uočimo nekoliko karakteristika ovog načina smeštanja. Koren je smešten na poziciji 1. Elementi polaznog niza nalaze se na pozicijama $[n, 2n-1]$. Element koji se u polaznom nizu nalazi na poziciji p, se u segmentnom drvetu nalazi na poziciji $p+n$. Levo dete čvora k nalazi se na poziciji $2k$, a desno na poziciji $2k+1$. Dakle, na parnim pozicijama se nalaze leva deca svojih roditelja, a na neparnim desna. Roditelj čvora k nalazi se na poziciji $\lfloor \frac{k}{2} \rfloor$.

Formiranje segmentnog drveta na osnovu datog niza je veoma jednostavno. Prvo se elementi polaznog niza prekopiraju u drvo, krenuvši od pozicije n. Zatim se svi unutrašnji čvorovi drveta (od pozicije $n-1$, pa unazad do pozicije 1) popunjavaju kao zbirovi svoje dece (na poziciju k upisujemo zbir elemenata na pozicijama $2k$ i $2k+1$).

```
// na osnovu datog niza a dužine n
// u kom su elementi smešteni od pozicije 0
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije 1
void formirajSegmentnoDrvo(int a[], int n, int drvo[]) {
  // kopiramo originalni niz u listove
  copy_n(a, n, drvo + n);
  // ažuriramo roditelje već upisanih elemenata
  for (int k = n-1; k >= 1; k--)
    drvo[k] = drvo[2*k] + drvo[2*k+1];
}
```

Složenost ove operacije je očigledno linearna u odnosu na dužinu niza n.

Prethodni pristup formira drvo odozdo naviše (prvo se popune listovi, pa onda koren). Još jedan način je da se drvo formira rekurzivno, odozgo naniže. Iako je ova implementacija komplikovanija i malo neefikasnija, pristup odozgo naniže je u nekim kasnijim operacijama neizbežan, pa ga ilustrujemo na ovom jednostavnom primeru. Svaki čvor drveta predstavlja zbir određenog segmenta pozicija. Segment je jednoznačno određen pozicijom k u nizu, ali da bismo olakšali implementaciju granice tog segmenta možemo kroz rekurziju prosleđivati kao parametar funkcije, zajedno sa vrednošću k (neka je to segment $[x, y]$). Drvo krećemo da gradimo od korena gde je $k = 1$ i $[x, y] = [0, n-1]$. Ako roditeljski čvor pokriva segment $[x, y]$, tada levo dete pokriva segment $[x, \lfloor \frac{x+y}{2} \rfloor]$, a desno dete pokriva segment $[\lfloor \frac{x+y}{2} \rfloor + 1, y]$. Drvo popunjavamo rekurzivno, tako što prvo popunimo levo poddrvo, zatim desno i na kraju vrednost u korenu izračunavamo kao zbir vrednosti u levom i desnom detetu. Izlaz iz rekurzije predstavljaju listovi, koje prepoznajemo po tome što pokrivaju segmente dužine 1, i u njih samo kopiramo elemente sa odgovarajućih pozicija polaznog niza.

```
// od elemenata niza a sa pozicija [x, y]
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije k
void formirajSegmentnoDrvo(int a[], int drvo[], int k, int x, int y) {
   if (x == y)
      // u listove prepisujemo elemente polaznog niza
      drvo[k] = a[x];
   else {
      // rekurzivno formiramo levo i desno poddrvo
      int s = (x + y) / 2;
      formirajSegmentnoDrvo(a, drvo, 2*k, x, s);
      formirajSegmentnoDrvo(a, drvo, 2*k+1, s+1, y);
      // izračunavamo vrednost u korenu
```

```
      drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
}

// na osnovu datog niza a dužine n
// u kom su elementi smešteni od pozicije 0
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije 1
void formirajSegmentnoDrvo(int a[], int n, int drvo[]) {
  // krećemo formiranje od korena koji se nalazi u nizu drvo
  // na poziciji 1 i pokriva elemente na pozicijama [0, n-1]
  formirajSegmentnoDrvo(a, drvo, 1, 0, n-1);
}
```

Razmotrimo sada kako bismo našli zbir elemenata na pozicijama iz segmenta $[2, 6]$, tj. zbir elemenata $1, 2, 6, 5, 1$. U segmentnom drvetu taj segment je smešten na pozicijama $[2 + 8, 6 + 8] = [10, 14]$. Zbir prva elementa $(1, 2)$ se nalazi u čvoru iznad njih, zbir naredna dva elemenata $(6, 5)$ takođe, dok se u roditeljskom čvoru elementa 1 nalazi njegov zbir sa elementom 4, koji ne pripada segmentu koji sabiramo. Zato zbir elemenata pozicija $[10, 14]$ možemo razložiti na zbir elemenata na pozicijama $[5, 6]$ i elementa na poziciji 14.

Razmotrimo i kako bismo računali zbir elemenata na pozicijama iz segmenta $[3, 7]$, tj. zbir elemenata $2, 6, 5, 1, 4$. U segmentnom drvetu taj segment je smešten na pozicijama $[3 + 8, 7 + 8] = [11, 15]$. U roditeljskom čvoru elementa 2 nalazi se njegov zbir sa elementom 1 koji ne pripada segmentu koji sabiramo. Zbirovi elemenata 6 i 5 i elemenata 1 i 4 se nalaze u čvorovima iza njih.

Generalno, za sve unutrašnje elemente segmenta smo sigurni da se njihov zbir nalazi u čvorovima iznad njih. Jedini izuzetak mogu da budu elementi na krajevima segmenta. Ako je element na levom kraju segmenta levo dete (što je ekvivalentno tome da se nalazi na parnoj poziciji) tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom desno od njega koji takođe pripada segmentu koji treba sabrati (osim eventualno u slučaju jednočlanog segmenta). U suprotnom (ako se nalazi na neparnoj poziciji), u njegovom roditeljskom čvoru je njegov zbir sa elementom levo od njega, koji ne pripada segmentu koji sabiramo. U toj situaciji, taj element ćemo posebno dodati na zbir i isključiti iz segmenta koji sabiramo pomoću roditeljskih čvorova. Ako je element na desnom kraju segementa levo dete (ako se nalazi na parnoj poziciji), tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom desno od njega, koji ne pripada segmentu koji sabiramo. I u toj situaciji, taj element ćemo posebno dodati na zbir i isključiti iz segmenta koji sabiramo pomoću roditeljskih čvorova. Na kraju, ako se krajnji desni element nalazi u desnom čvoru (ako je na neparnoj poziciji), tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom levo od njega, koji pripada segmentu koji sabiramo (osim eventualno u slučaju jednočlanog segmenta).

```
// izračunava se zbir elemenata polaznog niza dužine n koji se
```

```
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drveta
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
int saberi(int drvo[], int n, int a, int b) {
   a += n; b += n;
   int zbir = 0;
   while (a <= b) {
      if (a % 2 == 1) zbir += drvo[a++];
      if (b % 2 == 0) zbir += drvo[b--];
      a /= 2;
      b /= 2;
   }
   return zbir;
}
```

Pošto se u svakom koraku dužina segmenta $[a, b]$ polovi, a ona je u početku sigurno manja ili jednaka n, složenost ove operacije je $O(\log n)$.

Prethodna implementacija vrši izračunavanje odozdo naviše. I za ovu operaciju možemo napraviti i rekurzivnu implementaciju koja vrši izračunavanje odozgo naniže. Za svaki čvor u segmentnom drvetu funkcija vraća koliki je doprinos segmenta koji odgovara tom čvoru i njegovim naslednicima traženom zbiru elemenata na pozicijama iz segmenta $[a, b]$. Na počeku krećemo od korena i računamo doprinos celog drveta zbiru elemenata iz segmenta $[a, b]$. Postoje tri različita moguća odnosa između segmenta $[x, y]$ i segmenta $[a, b]$. Ako su disjunktni, doprinos tekućeg čvora zbiru segmenta $[a, b]$ je nula. Ako je $[x, y]$ u potpunosti sadržan u $[a, b]$, tada je doprinos potpun, tj. ceo zbir segmenta $[x, y]$ (a to je broj upisan u nizu na poziciji k) doprinosi zbiru elemenata na pozicijama iz segmenta $[a, b]$. Na kraju, ako se segmenti seku, tada je doprinos tekućeg čvora jednak zbiru doprinosa njegovog levog i desnog deteta. Odatle sledi naredna implementacija.

```
// izračunava se zbir onih elemenata polaznog niza koji se
// nalaze na pozicijama iz segmenta [a, b] koji se nalaze u
// segmentnom drvetu koje čuva elemente polaznog niza koji se
// nalaze na pozicijama iz segmenta [x, y] i smešteno je u nizu
// drvo od pozicije k
int saberi(int drvo[], int k, int x, int y, int a, int b) {
  // segmenti [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // segment [x, y] je potpuno sadržan unutar segmenta [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // segmenti [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return saberi(drvo, 2*k, x, s, a, b) +
         saberi(drvo, 2*k+1, s+1, y, a, b);
}

// izračunava se zbir elemenata polaznog niza dužine n koji se
```

```
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drveta
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
int saberi(int drvo[], int n, int a, int b) {
  // krećemo od drveta smeštenog od pozicije 1 koje
  // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
  return saberi(drvo, 1, 0, n-1, a, b);
}
```

Iako nije sasvim očigledno i ova implementacija će imati složenost $O(\log n)$.

Prilikom ažuriranja nekog elementa potrebno je ažurirati sve čvorove na putanji od tog lista do korena. S obzirom da znamo poziciju roditelja svakog čvora i ova operacija se može veoma jednostavno implementirati.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije 1
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void promeni(int drvo[], int n, int i, int v) {
   // prvo ažuriramo odgovarajući list
   int k = i + n;
   drvo[k] = v;
   // ažuriramo sve roditelje izmenjenih čvorova
   for (k /= 2; k >= 1; k /= 2)
      drvo[k] = drvo[2*k] + drvo[2*k+1];
}
```

Pošto se k polovi u svakom koraku petlje, a kreće od vrednosti najviše $2n - 1$, i složenost ove operacije je $O(\log n)$.

I ovu operaciju možemo implementirati odozgo naniže.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije k
// koje sadrži elemente polaznog niza a dužine n sa pozicija iz
// segmenta [x, y], nakon što se na poziciju i niza upiše vrednost v
void promeni(int drvo[], int k, int x, int y, int i, int v) {
   if (x == y)
     // ažuriramo vrednost u listu
     drvo[k] = v;
   else {
     // proveravamo da li se pozicija i nalazi levo ili desno
     // i u zavisnosti od toga ažuriramo odgovarajuće poddrvo
     int s = (x + y) / 2;
     if (x <= i && i <= s)
       promeni(drvo, 2*k, x, s, i, v);
     else
       promeni(drvo, 2*k+1, s+1, y, i, v);
     // pošto se promenila vrednost u nekom od dva poddrveta
     // moramo ažurirati vrednost u korenu
     drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
```

```
}

// ažurira segmentno drvo smešteno u niz drvo od pozicije 1
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void promeni(int drvo[], int n, int i, int v) {
   // krećemo od drveta smeštenog od pozicije 1 koje
   // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
   promeni(drvo, 1, 0, n-1, i, v);
}
```

Složenost je opet $O(\log n)$, jer se dužina intervala $[x, y]$ u svakom koraku bar dva puta smanjuje.

Umesto funkcije `promeni` često se razmatra funkcija `uvecaj` koja element na poziciji `i` polaznog niza uvećava za datu vrednost `v` i u skladu sa tim ažurira segmentno drvo. Svaka od ove dve funkcije se lako izražava preko one druge.

Implementacija segmentnog drveta za druge asocijativne operacije je skoro identična, osim što se operator `+` menja drugom operacijom.

Fenvikova drveta (BIT)

U nastavku ćemo razmotriti *Fenvikova drveta* tj. *binarno indeksirana drveta* (engl. binary indexed tree, BIT) koja koriste malo manje memorije i mogu biti za konstantni faktor brža od segmentnih drveta (iako je složenost operacija asimptotski jednaka). Sa druge strane, za razliku od segmentnih drveta koja su pogodna za različite operacije, Fenvikova drveta su specijalizovana samo za asocijativne operacije koje imaju inverz (npr. zbirovi ili proizvodi elemenata segmenata se mogu nalaziti uz pomoć BIT, ali ne i minimumi, nzd-ovi i slično). Segmentna drveta mogu da urade sve što i Fenvikova, dok obratno ne važi.

Iako se naziva drvetom, Fenvikovo drvo zapravo predstavlja niz vrednosti zbirova nekih pametno izabranih segmenata. Izbor segmenata je u tesnoj vezi sa binarnom reprezentacijom indeksa. Ponovo ćemo jednostavnosti radi pretpostaviti da se vrednosti u nizu smeštaju od pozicije 1 (vrednost na poziciji 0 je irelevantna) i to i u polaznom nizu i u nizu u kom se smešta drvo. Prilagođavanje koda situaciji u kojoj su u polaznom nizu elementi smešteni od pozicije nula, veoma je jednostavno (samo je na početku svake funkcije koja radi sa drvetom indeks polaznog niza potrebno uvećati za jedan pre dalje obrade). Ako je polazni niz dužine n, elementi drveta će se smeštati u poseban niz na pozicije $[1, n]$.

Ključna ideja Fenvikovog drveta je sledeća: *u drvetu se na poziciji k čuva zbir vrednosti polaznog niza iz segmenta pozicija oblika $(f(k), k]$ gde je $f(k)$ broj koji se dobije od broja k tako što se iz binarnog zapisa broja k obriše prva jedinica sdesna.*

Na primer, na mestu $k = 21$ zapisuje se zbir elemenata polaznog niza na pozicijama iz intervala $(20, 21]$, jer se broj 21 binarno zapisuje kao 10101 i brisanjem jedinice dobija se binarni zapis 10100 tj. broj 20 (važi da je $f(21) = 20$). Na poziciji broj 20 nalazi se zbir elemenata sa pozicija iz intervala $(16, 20]$, jer se brisanjem jedinice dobija binarni zapis 10000 tj. broj 16 (važi da je $f(20) = 16$). Na poziciji 16 se čuva zbir elemenata sa pozicija iz intervala $(0, 16]$, jer se brisanjem jedinice iz binarnog zapisa broja 16 dobija 0 (važi da je $f(16) = 0$).

Za niz $3, 4, 1, 2, 6, 5, 1, 4$, Fenvikovo drvo bi čuvalo sledeće vrednosti.

```
0    1      2     3     4     5     6     7     8  k
     1     10    11   100   101   110   111  1000  k  binarno
     0      0    10     0   100   100   110     0  f(k) binarno
 (0,1]  (0,2] (2,3] (0,4] (4,5] (4,6] (6,7] (0,8]  interval
     3      4     1     2     6     5     1     4  niz
     3      7     1    10     6    11     1    26  drvo
```

Nadovezivanjem intervala $(0, 16]$, $(16, 20]$ i $(20, 21]$ dobija se interval $(0, 21]$ tj. prefiks niza do pozicije 21. Zbir elemenata u prefiksu se, dakle, može se dobiti kao zbir nekoliko elemenata zapisanih u Fenvikovom drvetu. Ovo, naravno, važi za proizvoljni indeks (ne samo za 21). Broj elemenata čijim se sabiranjem dobija zbir prefiksa je samo $O(\log n)$. Naime, u svakom koraku se broj broj jedinica u binarnom zapisu tekućeg indeksa smanjuje, a broj n se zapisuje sa najviše $O(\log n)$ binarnih jedinica.

Implementacija je veoma jednostavna, kada se pronađe način da se iz binarnog zapisa broja ukloni prva jedinica sleva tj. da se za dati broj k izračuna $f(k)$. Pod pretpostavkom da su brojevi zapisani u potpunom komplementu, izrazom `k & -k` može se dobiti broj koji sadrži samo jednu jedinicu i to na mestu poslednje jedinice u zapisu broja `k`. Oduzimanjem te vrednosti od broja `k` tj. izrazom `k - (k & -k)` dobijamo efekat brisanja poslednje jedinice u binarnom zapisu broja `k` i to predstavlja implementaciju funkcije f. Drugi način da se to uradi je da se izračuna vrednost `k & (k-1)`.

Zbir prefiksa $[0, k]$ polaznog niza možemo onda izračunati narednom funkcijom.

```
// na osnovu Fenvikovog drveta smeštenog u niz drvo
// izračunava zbir prefiksa (0, k] polaznog niza
int zbirPrefiksa(int drvo[], int k) {
  int zbir = 0;
  while (k > 0) {
     zbir += drvo[k];
     k -= k & -k;
  }
}
```

Kada znamo zbir prefiksa, zbir proizvoljnog segmenta $[a, b]$ možemo izračunati kao razliku zbira prefiksa $(0, b]$ i zbira prefiksa $(0, a - 1]$. Pošto se oba računaju u vremenu $O(\log n)$, i zbir svakog segmenta možemo izračunati u vremenu

$O(\log n)$.

Osnovna prednost Fenvikovih drveta u odnosu na niz svih zbirova prefiksa je to što se mogu efikasno ažurirati. Razmotrimo funkciju koja ažurira drvo nakon uvećanja elementa u polaznom nizu na poziciji k za vrednost x. Tada je za x potrebno uvećati sve one zbirove u drvetu u kojima se kao sabirak javlja i element na poziciji k. Ti brojevi se izračunavaju veoma slično kao u prethodnoj funkciji, jedino što se umesto oduzimanja vrednosti `k & -k` broj `k` u svakom koraku uvećava za `k & -k`.

```
// Ažurira Fenvikovo drvo smešteno u niz drvo nakon što se
// u originalnom nizu element na poziciji k uveća za x
void uvećaj(int drvo[], int n, int k, int x) {
  while (k <= n) {
    drvo[k] += x;
    k += k & -k;
  }
}
```

Objasnimo i dokažimo korektnost prethodne implementacije. Potrebno je ažurirati sve one pozicije m čiji pridruženi segment sadrži vrednost k, tj. sve one pozicije m takve da je $k \in (f(m), m]$, tj. $f(m) < k \leq m$. Ovo nikako ne može da važi za brojeve $m < k$, a sigurno važi za broj $m = k$, jer je $f(k) < k$, kada je $k > 0$ (a mi pretpostavljamo da je $1 \leq k \leq n$). Za brojeve $m > k$, sigurno važi desna nejednakost i potrebno je utvrditi da važi leva. Neka je $g(k)$ broj koji se dobija od k tako što se k sabere sa brojem koji ima samo jednu jedinicu u svom binarnom zapisu i to na poziciji na kojoj se nalazi poslednja jedinica u binarnom zapisu broja k. Na primer, za broj $k = 101100$, broj $g(k) = 101100 + 100 = 110000$. U implementaciji se broj $g(k)$ lako može izračunati kao `k + (k & -k)`. Tvrdimo da je najmanji broj m koji zadovoljava uslov $f(m) < k < m$ upravo $g(k)$. Zaista, očigledno važi $k < g(k)$ i $g(k)$ ima sve nule od pozicije poslednje jedinice u binarnom zapisu broja k (uključujući i nju), pa do kraja, pa se brisanjem njegove poslednje jedinice tj. izračunavanjem $f(g(k))$ sigurno dobija broj koji je strogo manji od k. Nijedan broj m između k i $g(k)$ ne može da zadovolji uslov da je $f(m) < k$. Naime, svi ti brojevi se poklapaju sa brojem k na svim pozicijama pre krajnjih nula, a na pozicijama krajnjih nula broja k imaju bar neku jedinicu, čijim se brisanjem dobija broj koji je veći ili jednak k. Po istom principu zaključujemo da naredni traženi broj mora biti $g(g(k))$, zatim $g(g(g(k)))$ itd. sve dok se ne dobije neki broj koji prevazilazi n. Zaista, važi da je $k < g(k) < g(g(k))$. Važi da je $f(g(g(k))) < f(g(k)) < k$, pa $g(g(k))$ zadovoljava uslov. Nijedan broj između $g(k)$ i $g(g(k))$ ne može da zadovolji uslov, jer se svi oni poklapaju sa $g(k)$ u svim binarnim ciframa, osim na njegovim krajnjim nulama gde imaju neke jedinice. Brisanjem poslednje jedinice se dobija broj koji je veći ili jednak $g(k)$, pa dobijeni broj ne može biti manji od k. Otuda sledi da su jedine pozicije koje treba ažurirati upravo pozicije iz serije k, $g(k)$, $g(g(k))$ itd., sve dok su one manje ili jednak n, pa je naša implementacija korektna.

Ostaje još pitanje kako u startu formirati Fenvikovo drvo, međutim, formiranje se može svesti na to da se kreira drvo popunjeno samo nulama, a da se zatim uvećava vrednost jednog po jednog elementa niza prethodnom funkcijom.

```
// Na osnovu niza a u kom su elementi smešteni
// na pozicijama iz segmenta [1, n] formira Fenvikovo drvo
// i smešta ga u niz drvo (na pozicije iz segmenta [1, n])
void formirajDrvo(int drvo[], int n, int a[]) {
  fill_n(a+1, n, 0);
  for (int k = 1; k <= n; k++)
     dodaj(drvo, n, k, a[k]);
}
```

Broj inverzija pomoću Fenvikovog drveta

Prikažimo sada upotrebu Fenvikovog drveta u rešavanju jednog algoritamskog problema. Umesto Fenvikovog, moglo je biti upotrebljeno i segmentno drvo.

Problem: Odredi koliko različitih parova elemenata u nizu je takvo da je prvi element strogo veći drugog. Na primer, u nizu $5, 4, 3, 1, 2$ takvi su parovi $(5, 4)$, $(5, 3)$, $(5, 1)$, $(5, 2)$, $(4, 3)$, $(4, 1)$, $(4, 2)$, $(3, 1)$ i $(3, 2)$ i ima ih 9.

Rešenje grubom silom podrazumeva proveru svakog para elemenata i očigledno je složenosti $O(n^2)$. Možemo i efikasnije od toga.

Osnovna ideja je da radimo induktivno i da obrađujemo niz element po element. Pretpostavimo da smo ojačali induktivnu hipotezu i da za obrađeni prefiks niza znamo ne samo broj inverznih parova u tom prefiksu, već i frekvencije svih elemenata (ako su svi elementi u nizu različiti frekvencije će biti ili 0 ili 1). Da bismo odredili koliko inverzija tekući element pravi sa elementima ispred sebe, potrebno je da znamo koliko postoji elemenata koji su se javili pre tekućeg, a koji su strogo veći od njega. To možemo otkriti tako što saberemo frekvencije svih elemenata koji su strogo veći od tekućeg (ako je tekući element a_i, zanima nas zbir elemenata niza frekvencija koji se javljaju na pozicijama iz intervala $[a_i, max]$, gde je max najveći element niza. To možemo saznati jednostavno iz Fenvikovog drveta kao razliku zbirova dva prefiksa.

Primećujemo da su segmenti čiji nas zbirovi zanimaju stalno sufiksi niza frekvencija. Ovo nas može asocirati na to da bismo obilaskom u suprotnom smeru mogli malo popraviti efikasnost, jer bismo umesto zbirova sufiksa koji zahtevaju dva obilaska izračunavali zbirove prefiksa koji zahtevaju samo jedan obilazak. Zaista, ako niz obrađujemo unatrag i računamo frekvencije svih ranije viđenih elemenata, onda inverzije u kojima učestvuje tekući element i oni elementi iza njega određujemo tako što prebrojimo koliko ima viđenih elemenata koji su strogo manji od tekućeg, a to možemo saznati ako saberemo frekvencije koje se nalaze na pozicijama iz intervala $[1, a - 1]$. Na kraju ne smemo zaboraviti da održimo induktivnu hipotezu tako što nakon obrade uvećamo frekvenciju pojavljivanja tekućeg elementa.

Prethodni pristup nije moguć ako se u nizu javljaju negativni elementi, a prilično je memorijski neefikasan ako je maksimalni element niza veliki (memorijska složenost BIT-a je $O(max)$). Nama nisu relevantne same vrednosti, već samo njihov međusobni odnos. Zato pre obrade pomoću BIT-a možemo uraditi takozvanu kompresiju indeksa (engl. index compression). Svaki element u nizu ćemo zameniti njegovom pozicijom u sortiranom redosledu (krenuvši od 1). Iste elemente možemo zameniti istim pozicijama. Najlakši način da se to postigne je da se napravi sortirana kopija niza, i zatim da za se svaki element polaznog niza binarnom pretragom pronađe pozicija njegovog prvog pojavljivanja u sortiranoj kopiji (u jeziku C++ to možemo jednostavno postići bibliotečkim funkcijama `sort` i `lower_bound`).

```
// operacije za rad sa Fenvikovim drvetom

void dodaj(vector<int>& drvo, int k, int v) {
  while (k < drvo.size()) {
    drvo[k] += v;
    k += k & -k;
  }
}

int zbirPrefiksa(const vector<int>& drvo, int k) {
  int zbir = 0;
  while (k > 0) {
    zbir += drvo[k];
    k -= k & -k;
  }
  return zbir;
}

// izračunava broj parova i < j takvih da je a[i] > a[j]
int brojInverzija(const vector<int>& a) {
  // sortiramo niz
  int n = a.size();
  vector<int> a_sort = a;
  sort(begin(a_sort), end(a_sort));
  // Fenvikovo drvo
  vector<int> drvo(n+1, 0);
  // broj inverzija
  int broj = 0;
  for (int i = n-1; i >= 0; i--) {
    // na osnovu sortiranog niza a,
    // određujemo koji je po veličini u nizu a element a[i] -
    // broji se od 1
    auto it = lower_bound(begin(a_sort), end(a_sort), a[i]);
    int x = distance(begin(a_sort), it) + 1;
    // uvećavamo broj inverzija za broj do sada viđenih elemenata
    // koji su strogo manji od x
    broj += zbirPrefiksa(drvo, x-1);
```

```
    // ažuriramo frekvenciju pojavljivanja elementa x
    dodaj(drvo, x, 1);
  }
  // vraćamo ukupan broj inverzija
  return broj;
}
```

Sortiranje nosi složenost $O(n \log n)$. Nakon toga se za svaki od n elemenata izvršava jedna binarna pretraga čija je složenost $O(\log n)$, jedno izračunavanje zbira prefiksa Fenvikovog drveta čija je složenost takođe $O(\log n)$ i na kraju jedno ažuriranje vrednosti elementa u originalnom nizu i Fenvikovom drvetu čija je složenost takođe $O(\log n)$, tako da je ukupna složenost $O(n \log n)$.

Rešenje zasnovano na principu dekompozicije (prilagođenoj varijanti sortiranja objedinjavanjem) iste složenosti prikazaćemo kasnije.

Ažuriranje celih raspona niza odjednom

Videli smo da i segmentna i Fenvikova drveta podržavaju efikasno izračunavanje statistika određenih segmenata (raspona) niza i ažuriranje pojedinačnih elemenata niza. Ažuriranje celih segmenata niza odjednom nije direktno podržano. Ako se ono svede na pojedinačno ažuriranje svih elemenata unutar segmenta, dobija se loša složenost.

Moguće je jednostavno upotrebiti Fenvikovo drvo tako da se efikasno podrži uvećavanje svih elemenata iz datog segmenta odjednom, ali onda se gubi mogućnost efikasnog izračunavanja zbirova elemenata segmenata, već je samo moguće efikasno vraćati vrednosti pojedinačnih elemenata niza. Osnovna ideja je da se održava niz razlika susednih elemenata polaznog niza i da se taj niz razlika čuva u Fenvikovom drvetu. Uvećavanje svih elemenata segmenata polaznog niza za neku vrednost x, svodi se na promenu dva elementa niza razlika, dok se rekonstrukcija elementa polaznog niza na osnovu niza razlika svodi na izračunavanje zbira odgovarajućeg prefiksa, što se pomoću Fenvikovog drveta može uraditi veoma efikasno. Na ovaj način i ažuriranje celih segmenata niza odjednom i očitavanje pojedinačnih elemenata možemo postići u složenosti $O(\log n)$, što nije bilo moguće samo uz korišćenje niza razlika (uvećavanja svih elemenata nekog segmenta je tada bilo složenosti $O(1)$, ali je očitavanje vrednosti iz niza bilo složenosto $O(n)$).

Efikasno uvećanje svih elemenata u datom segmentu za istu vrednost i izračunavanje zbirova segmenata moguće je implementirati pomoću održavanja dva Fenvikova drveta (čime se nećemo baviti).

Ove operacije se mogu implementirati nad segmentnim drvetima ako se primeni takozvana tehnika *lenje propagacije* (engl. lazy propagation). Za lenju propagaciju nam je bitno da znamo rad sa segmentnim drvetom odozgo naniže.

Svaki čvor u segmentnom drvetu se odnosi na određeni segment elemenata polaznog niza i čuva zbir tog segmenta. Ako se taj segment u celosti sadrži unutar segmenta koji se ažurira, možemo unapred izračunati za koliko se povećava vrednost u korenu. Naime, pošto se svaka vrednost u segmentu povećava za v, tada se vrednost zbira tog segmenta povećava za $k \cdot v$, gde je k broj elemenata u tom segmentu. Vrednost zbira u korenu time biva ažurirana u konstantnom vremenu, ali vrednosti zbirova unutar poddrveta kojima je to koren (uključujući i vrednosti u listovima koje odgovaraju vrednostima polaznog niza) i dalje ostaju neažurne. Njihovo ažuriranje zahtevalo bi linearno vreme, što nam je nedopustivo. Ključna ideja je da se ažuriranje tih vrednosti odloži i da se one ne ažuriraju odmah, već samo tokom neke kasnije posete tim čvorovima, do koje bi došlo i inače (ne želimo da te čvorove posećujemo samo zbog ovog ažuriranja, već ćemo ažuriranje uraditi usput, tokom neke druge posete tim čvorovima koja bi se svakako morala desiti). Postavlja se pitanje kako da signaliziramo da vrednosti zbirova u nekom poddrvetu nisu ažurne i dodatno ostavimo uputstvo na koji način se mogu ažurirati. U tom cilju proširujemo čvorove i u svakom od njih pored vrednosti zbira segmenta čuvamo i dodatni *koeficijente lenje propagacije*. Ako drvo u svom korenu ima koeficijent lenje propagacije c koji je različit od nule, to znači da vrednosti zbirova u celom tom drvetu nisu ažurne i da je svaki od listova tog drveta potrebno povećati za c i u odnosu na to ažurirati i vrednosti zbirova u svim unutrašnjim čvorovima tog drveta (uključujući i koren). Ažuriranje se može odlagati sve dok vrednost zbira u nekom čvoru ne postane zaista neophodna, a to je tek prilikom upita izračunavanja vrednosti zbira nekog segmenta. Ipak, vrednosti zbirova u čvorovima ćemo ažurirati i češće i to zapravo prilikom svake posete čvoru - bilo u sklopu operacije uvećanja vrednosti iz nekog segmenta pozicija polaznog niza, bilo u sklopu upita izračunavanja zbira nekog segmenta. Na početku obe rekurzivne funkcije ćemo proveravati da li je vrednost koeficijenta lenje propagacije različita od nule i ako jeste, ažuriraćemo vrednost zbira tako što ćemo ga uvećati za proizvod tog koeficijenta i broja elemenata u segmentu koji taj čvor predstavlja, a zatim koeficijente lenje propagacije oba njegova deteta uvećati za taj koeficijent (time koren drveta koji trenutno posećujemo postaje ažuran, a njegovim poddrvetima se daje uputstvo kako ih u budućnosti ažurirati). Primetimo da se izbegava ažuriranje celog drveta odjednom, već se ažurira samo koren, što je operacija složenosti $O(1)$.

Imajući ovo u vidu, razmotrimo kako se može implementirati funkcija koja vrši uvećanje svih elemenata nekog segmenata. Njena invarijanta će biti da su svi čvorovi u drvetu koje ili sadržati ažurne vrednosti zbirova ili će biti ispravno obeleženi za kasnija ažuriranja (preko koeficijenta lenje propagacije), a da će nakon njenog izvršavanja koren drveta na kom je pozvana sadržati aktuelnu vrednost zbira. Nakon početnog obezbeđivanja da vrednost u tekućem čvoru postane ažurna, moguća su tri sledeća slučaja. Ako je segment u tekućem čvoru disjunktan u odnosu na segment koji se ažurira, tada su svi čvorovi u poddrvetu kojem je on koren već ili ažurni ili ispravno obeleženi za kasnije ažuriranje i nije potrebno ništa uraditi. Ako je segment koji odgovara tekućem čvoru potpuno sadržan u segmentu čiji se elementi uvećavaju, tada se njegova vrednost ažurira

(uvećavanjem za $k \cdot v$, gde je k broj elemenata segmenta koji odgovara tekućem čvoru, a v vrednost uvećanja), a njegovoj deci se koeficijent lenje propagacije uvećava za v. Na kraju, ako se dva segmenta seku, tada se prelazi na rekurzivnu obradu dva deteta. Nakon izvršavanja funkcije nad njima, sigurni smo da će svi čvorovi u levom i desnom poddrvetu zadovoljavati uslov invarijante i da će oba korena imati ažurne vrednosti. Ažurnu vrednost u korenu postizaćemo sabiranjem vrednosti dva deteta.

```
// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [x, y]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void promeni(int drvo[], int lenjo[], int k, int x, int y,
             int a, int b, int v) {
   // ažuriramo vrednost u korenu, ako nije ažurna
   if (lenjo[k] != 0) {
      drvo[k] += (y - x + 1) * lenjo[k];
      // ako nije u pitanju list propagaciju prenosimo na decu
      if (x != y) {
         lenjo[2*k] += lenjo[k];
         lenjo[2*k+1] += lenjo[k];
      }
      lenjo[k] = 0;
   }
   // ako su intervali disjunktni, ništa nije potrebno raditi
   if (b < x || y < a) return;
   // ako je interval [x, y] ceo sadržan u intervalu [a, b]
   if (a <= x && y <= b) {
     drvo[k] += (y - x + 1) * v;
     // ako nije u pitanju list propagaciju prenosimo na decu
     if (x != y) {
        lenjo[2*k] += v;
        lenjo[2*k+1] += v;
     }
   } else {
     // u suprotnom se intervali seku,
     // pa rekurzivno obilazimo poddrveta
     int s = (x + y) / 2;
     promeni(drvo, lenjo, 2*k, x, s, a, b, v);
     promeni(drvo, lenjo, 2*k+1, x, s, a, b, v);
     drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
}

// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [0, n-1]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
```

```
// segmenta [a, b] uvećani za vrednost v
void promeni(int drvo[], int lenjo[], int n,
             int a, int b, int v) {
   promeni(drvo, lenjo, 1, 0, n-1, a, b, v);
}
```

Prikažimo rad ove funkcije na jednom primeru.

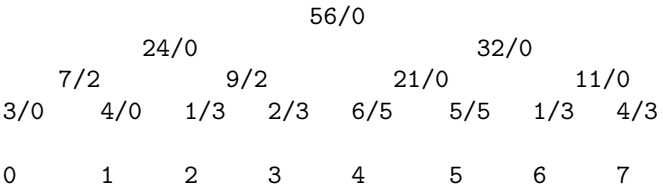
```
                    26/0
          10/0                    16/0
    7/0         3/0         11/0         5/0
3/0    4/0   1/0   2/0   6/0    5/0   1/0   4/0

0      1     2     3     4      5     6     7
```

Prikažimo kako bismo sve elemente iz segmenta pozicija $[2, 7]$ uvećali za 3. Segmenti $[0, 7]$ i $[2, 7]$ se seku, pa stoga ažuriranje prepuštamo naslednicima i nakon njihovog ažuriranja, pri povratku iz rekurzije vrednost određujemo kao zbir njihovih ažuriranih vrednosti. Na levoj strani se segment $[0, 3]$ seče sa $[2, 7]$ pa i on prepušta ažuriranje naslednicama i ažurira se tek pri povratku iz rekurzije. Segment $[0, 1]$ je disjunktan u odnosu na $[2, 7]$ i tu onda nije potrebno ništa raditi. Segment $[2, 3]$ je ceo sadržan u $[2, 7]$, i kod njega direktno možemo da znamo kako se zbir uvećava. Pošto imamo dva elementa i svaki se uvećava za 3, zbir se uvećava ukupno za $2 \cdot 3 = 6$ i postavlja se na 9. U ovom trenutku izbegavamo ažuriranje svih vrednosti u drvetu u kom je taj element koren, već samo naslednicima upisujemo da je potrebno propagirati uvećanje za 3, ali samu propagaciju odlažemo za trenutak kada ona postane neophodna. U povratku iz rekurzije, vrednost 10 uvećavamo na $7 + 9 = 16$. Što se tiče desnog poddrveta, segment $[4, 7]$ je ceo sadržan u segmentu $[2, 7]$, pa i tu možemo izračunati vrednost zbira. Pošto se 4 elementa uvećavaju za po 3, ukupan zbir se uvećava za $4 \cdot 3 = 12$. Zato se vrednost 16 menja u 28. Propagaciju ažuriranja kroz poddrvo odlažemo i samo njegovoj deci beležimo da je uvećanje za 3 potrebno izvršiti u nekom kasnijem trenutku. Pri povratku iz rekurzije vrednost u korenu ažuriramo sa 26 na $16 + 28 = 44$. Nakon toga dobija se sledeće drvo.

```
                    44/0
          16/0                    28/0
    7/0         9/0         11/3         5/3
3/0    4/0   1/3   2/3   6/0    5/0   1/0   4/0

0      1     2     3     4      5     6     7
```

Pretpostavimo da se sada elementi iz segmenta $[0, 5]$ uvećavaju za 2. Ponovo se kreće od vrha i kada se ustanovi da se segment $[0, 7]$ seče sa $[0, 5]$ ažuriranje se prepušta deci i vrednost se ažurira tek pri povratku iz rekurzije. Segment $[0, 3]$ je ceo sadržan u $[0, 5]$, pa se zato vrednost 16 uvećava za $4 \cdot 2 = 8$ i postavlja na 24. Poddrveta se ne ažurira odmah, već se samo njihovim korenima upisuje da je sve vrednosti potrebno ažurirati za 2. U desnom poddrvetu segment $[4, 7]$

se seče sa $[0, 5]$, pa se rekurzivno obrađuju poddrveta. Pri obradi čvora 11, primećuje se da je on trebao da bude ažuriran, međutim, još nije, pa se onda njegova vrednost ažurira i uvećava za $2 \cdot 3$ i sa 11 menja na 17. Njegovi naslednici se ne ažuriraju odmah, već samo ako to bude potrebno i njima se samo upisuje lenja vrednost 3. Tek nakon toga, se primećuje da se segment $[4, 5]$ ceo sadrži u $[0, 5]$, pa se onda vrednost 17 uvećava za $2 \cdot 2 = 4$ i postavlja na 21. Poddrveta se ne ažuriraju odmah, već samo po potrebi tako što se u njihovim korenima postavi vrednost lenjog koeficijenta. Pošto je u njima već upisana vrednost 3, ona se sada uvećava za 2 i postavlja na 5. Prelazi se tada na obradu poddrveta u čijem je korenu vrednost 5 i pošto ono nije ažurno, prvo se ta vrednost 5 uvećava za $2 \cdot 3 = 6$ i postavlja na 11, a njegovoj deci se lenji koeficijent postavlja na 3. Nakon toga se primećuje da je segment $[6, 7]$ disjuktan sa $[0, 5]$ i ne radi se ništa. U povratku kroz rekurziju se ažuriraju vrednosti roditeljskih čvorova i dolazi se do narednog drveta.



Funkcija izračunavanja vrednosti zbira segmenta ostaje praktično nepromenjena, osim što se pri ulasku u svaki čvor vrši njegovo ažuriranje.

```
// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [x, y]
// izračunava se zbir elemenata polaznog niza sa pozicija
// iz segmenta [a, b]
int saberi(int drvo[], int lenjo[], int k, int x, int y,
           int a, int b) {
  // ažuriramo vrednost u korenu, ako nije ažurna
  if (lenjo[k] != 0) {
     drvo[k] += (y - x + 1) * lenjo[k];
     if (x != y) {
       lenjo[2*k] += lenjo[k];
       lenjo[2*k+1] += lenjo[k];
     }
     lenjo[k] = 0;
  }

  // intervali [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // interval [x, y] je potpuno sadržan unutar intervala [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // intervali [x, y] i [a, b] se seku
  int s = (x + y) / 2;
```

```
  return saberi(drvo, lenjo, 2*k, x, s, a, b) +
         saberi(drvo, lenjo, 2*k+1, s+1, y, a, b);
}

// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [0, n-1]
// izračunava se zbir elemenata polaznog niza sa pozicija
// iz segmenta [a, b]
int saberi(int drvo[], int lenjo[], int n, int a, int b) {
  // računamo doprinos celog niza,
  // tj. elemenata iz intervala [0, n-1]
  return saberi(drvo, lenjo, 1, 0, n-1, a, b);
}
```

Prikažimo sada rad prethodne funkcije na tekućem primeru. Razmotrimo kako se sada izračunava zbir elemenata u segmentu $[3, 5]$. Krećemo od vrha. Segment $[0, 7]$ se seče sa $[3, 5]$, pa se rekurzivno obrađuju deca. U levom poddrvetu segment $[0, 3]$ takođe ima presek sa $[3, 5]$ pa prelazimo na naredni nivo rekurzije. Prilikom posete čvora u čijem je korenu vrednost 7 primećuje se da njegova vrednost nije ažurna, pa se koristi prilika da se ona ažurira, tako što se uveća za $2 \cdot 2 = 4$, a naslednicima se lenji koeficijent uveća za 2. Pošto je segment $[0, 1]$ disjunktan sa $[3, 5]$, vraća se vrednost 0. Prilikom posete čvora u čijem je korenu vrednost 9 primećuje se da njegova vrednost nije ažurna, pa se koristi prilika da se ona ažurira, tako što se uveća za $2 \cdot 2 = 4$, a naslednicima se lenji koeficijent uveća za 2. Segment $[2, 3]$ se seče sa $[3, 5]$, pa se rekurzivno vrši obrada poddrveta. Vrednost 1 se prvo ažurira tako što se poveća za $1 \cdot 5 = 5$, a onda, pošto je $[2, 2]$ disjunktno sa $[3, 5]$ vraća se vrednost 0. Vrednost 2 se takođe prvo ažurira tako što se poveća za $1 \cdot 5 = 5$, a pošto je segment $[3, 3]$ potpuno sadržan u $[3, 5]$ vraća se vrednost 7. U desnom poddrvetu je čvor sa vrednošću 32 ažuran, segment $[4, 7]$ se seče sa $[3, 5]$, pa se prelazi na obradu naslednika. Čvor sa vrednošću 21 je ažuran, segment $[4, 5]$ je ceo sadržan u $[3, 5]$, pa se vraća vrednost 21. Čvor sa vrednošću 11 je takođe ažuran, ali je segment $[6, 7]$ disjunktan u odnosu na $[3, 5]$, pa se vraća vrednost 0. Dakle, čvorovi 13 i 24 vraćaju vrednost 7, čvor 32 vraća vrednost 21, pa čvor 56 vraća vrednost 28. Stanje drveta nakon izvršavanja upita je sledeće.

```
                        56/0
          24/0                          32/0
   11/0          13/0          21/0          11/0
3/2    4/2    6/0    7/0    6/5    5/5    1/3    4/3

0      1      2      3      4      5      6      7
```

Čas 7.1, 7.2, 7.3 - primena sortiranja i binarne pretrage

Binarna pretraga i sortiranje su podržani kroz bibliotečke funkcije skoro svih programskih jezika. U jeziku C++, koriste se sledeće funkcije.

- `sort` - 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. Postoji više načina da se to uradi, ali najčešće ćemo koristiti anonimnu (lambda) funkciju.
- Funkcija `binary_search` vrši binarnu pretragu niza i vraća logičku vrednost tačno ako i samo ako 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 funkcije `binary_search`.
- Funkcija `lower_bound` vraća iterator na prvi element koji je veći ili jednak datom. Parametri su isti kao i kod funkcije `binary_search`.
- Funkcija `upper_bound` vraća iterator na prvi element koji je strogo veći od datog. Parametri su isti kao i kod funkcije `binary_search`.

Primene sortiranja

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. Ovo rešenje je relativno komplikovano implementirati, a uz to je i veoma neefikasno (broj podskupova je $\binom{n}{k}$, što je $O(n^k)$).

Bolje i efikasnije rešenje se zasniva na sortiranju. Naime, kada se polazni paketi sortiraju po broju čokoladica, učenici treba da uzmu uzastopnih k paketa. Pret-

postavimo da nakon sortiranja imamo niz $a_0, a_1, \ldots, a_{n-1}$. Učenici treba da uzmu redom pakete od a_i, do a_{i+k-1}, za neko $0 \leq i \leq n-k$.

Dokažimo prethodnu činjenicu i formalno. Pretpostavimo suprotno, da paketi koji daju najmanji raspon ne čine uzastopan niz i da svaki uzastopni niz paketa dužine k ima strogo veći raspon od raspona skupa uzetih paketa. Neka je prvi uzeti paket a_i. Tada sigurno postoji neki paket a_j za $i < j < i+k$ koji nije uzet, a umesto njega je uzet neki paket $a_{j'}$ za neko $i+k \leq j' < n$. Neka je j' poslednji paket koji je uzet. Kada bi učenik koji je uzeo paket $a_{j'}$ zamenio taj paket za a_j raspon bi se sigurno smanjio ili bar ostao isti (jer bi poslednji uzeti paket tada bio neki paket $a_{j''}$, za $j'' < j'$, a pošto je niz sortiran neopadajuće, važi da je $a_{j''} \leq a_j$, pa i $a_{j''} - a_i \leq a_{j'} - a_i$. Daljim zamenema istog tipa možemo doći do toga da su svi uzeti paketi uzastopni, a da je raspon manji ili jednak polaznom, što je u kontradikciji sa tim da je raspon polaznog skupa uzetih paketa strogo manji od raspona bilo kojeg niza k uzastopnih paketa.

Na osnovu prethodnog jasno je da niz brojeva čokoladica u paketima treba najpre sortirati i zatim odrediti minimum razlika vrednosti $a_{i+k-1} - a_i$, za $0 \leq i \leq n-k$ (korišćenjem uobičajenog algoritma za nalaženje minimuma). Složenošću ovog algoritma dominira složenost koraka sortiranja, a ona je $O(n \log n)$, ako se koriste bibliotečke implementacije.

```
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 (pretpostavlja se da se nikoja dva pacijenta ne preklapaju). Kolika je najveća pauza koju doktor može imati u toku tog dana, između pregleda dva svoja pacijenta?

Da bi se zadatak mogao rešiti neophodno je sortirati pacijente po vremenu dolaska. Nakon toga 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. Pošto parova ima $\frac{n(n-1)}{2}$, složenost ovog pristupa bi bila $O(n^2)$.

Bolje rešenje se može dobiti ako se niz pre toga sortira. Naime, 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 (u neopadajuće sortiranom nizu važi da iz $j' < j < i$ sledi $a_i - a_j \leq a_i - a_{j'}$, jer iz sortiranosti i $j < j'$ sledi da je $a_j \leq a_{j'}$, kao i da iz $i < j < j'$ sledi da je $a_j - a_i \leq a_{j'} - a_i$, jer iz sortiranosti i $j < j'$ sledi $a_j \leq a_{j'}$). Zato je nakon sortiranja dovoljno proveriti sve razlike između susednih elemenata i odrediti najmanju od njih (ako ima više istih, određujemo poslednju). Za ovo koristimo algoritam određivanja najmanjeg elementa, dok sortiranje možemo najlakše izvršiti bibliotečkom funkcijom. Sortiranje zahteva $O(n \log n)$ operacija, dok je traženje minimuma složenosti $O(n)$, tako da je ukupno vreme ovog postupka $O(n \log n)$.

```
pair<int, int> najblizeSobe(vector<int>& sobe) {
  // sortiramo niz soba
  sort(begin(sobe), end(sobe));
  // trazimo poziciju sobe koja je najbliza svojoj narednoj
  int min = 0;
  for (int i = 1; i < n-1; i++)
    if (sobe[i+1] - sobe[i] <= sobe[min+1] - sobe[min])
      min = i;
  // soba na poziciji min i njoj naredna su najblize
  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. Na primer, za niz $4, 8, 1, -6, 9, 5, -9, 10, -1, 3, 0, 1, 2$ treba prikazati $-1, 0, 1, 2, 3, 4, 5$. Ako ima više takvih podskupova, prikazati prvi (onaj u kojem su brojevi najmanji).

Zadatak se može veoma jednostavno rešiti ako se niz prvo sortira i ako se iz njega uklone duplikati. Kada je niz sortiran i kada u njemu nema duplikata, pronalaženje traženog podskupa se svodi na pronalaženje najdužeg segmenta niza koji čine uzastopni brojevi.

```
int najduziPodskupUzastopnih(int a[], int n) {
  // sortiramo niz
  sort(a, next(a, n));

  // sažimamo niz ostavljanjem po jednog elementa iz svake serije
  // jednakih
  n = distance(a, unique(a, next(a, n)));

  // određujemo dužinu najduže serije uzastopnih celih brojeva
  int duzinaTekuceSerije = 1;
  int 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?

Početna važna opaska je da se broj ljudi koji su trenutno na bazenu 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 (vremena odlaska i dolaska) 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 najbrojnijiPresekIntervala(
       const vector<pair<int, int>>& intervali) {

    // broj intervala
    int n = intervali.size();
    // niz karakterističnih trenutaka
    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);
    }

    // sortiramo niz promena
    sort(begin(promene), end(promene));

    // broj ljudi koji su trenutno prisutni na bazenu
    int trenutnoPrisutno = 0;
    // maksimalni broj prisutnih ljudi na bazenu do sada
    int maksPrisutno = 0;
    // obrađujemo jedan po jedan trenutak
    int i = 0;
    while (i < n) {
       // obrađujemo sve promene koje su se desile u tekućem
       // trenutku
       int trenutak = promene[i].first;
       while (i < n && promene[i].first == trenutak)
          trenutnoPrisutno += promene[i++].second;
       // ažuriramo maksimum ako je to potrebno
       if (trenutnoPrisutno > maksPrisutno)
         maksPrisutno = trenutnoPrisutno;
    }

    // vraćamo konačni maksimum
    return maksPrisutno;
}
```

Ako se sortiranje radi leksikografski, tako da su svi događaji koji su se desili u istom trenutku sortirani tako da prvo idu oni koji su otišli, pa onda oni koji su došli, tada ne moramo da imamo unutrašnju petlju, jer će se broj prvo smanjivati, pa onda rasti i neće biti moguće da se dobije pogrešan rezultat zato što su dodati neki posetioci pre nego što je konstatovano da su neki otišli.

```
...
    // obrađujemo jedan po jedan trenutak
```

```
int i = 0;
while (i < n) {
   // obrađujemo jednu promenu
   trenutnoPrisutno += promene[i++].second;
   // ažuriramo maksimum ako je to potrebno
   if (trenutnoPrisutno > maksPrisutno)
     maksPrisutno = trenutnoPrisutno;
}
```

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).

Ideja rešenja ovog zadatka je bliska ideji prethodnog. 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, na sledeći način. U toku prolaza kroz sortirani niz, broj intervala koji se trenutno presecaju se uvećava za jedan kada se naiđe na početak intervala, a umanjuje za jedan kada se naiđe na kraj. Tačka u kojoj broj presecajućih intervala sa nule poraste na strogo pozitivan broj, pamti se kao početak ukrupnjenog intervala, čiji je kraj prva naredna tačka u kojoj broj presecajućih intervala padne ponovo na nulu. Prilikom formiranja svakog ukrupnjenog intervala, dužina pokrivača se povećava za dužinu tog intervala.

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

typedef vector<pair<double, double>> Intervali;

void ukrupniIntervale(const Intervali& ulazni, Intervali& izlazni) {
  // formiramo niz promena
  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);
  }

  // sortiramo niz promena leksikografski,
  // prvo po x koordinati, a zatim po promeni (izlazak pre ulaska)
  sort(begin(promene), end(promene));

  // da li je trenutno započet neki izlazni interval koji još
  // nije završen
```

```
  bool zapocetIzlazni = false;
  // pocetak trenutnog izlaznogIntervala (ako je započet)
  double xPocetakIzlaznog = 0;
  // broj ulaznih intervala koji obuhvataju trenutnu koordinatu x
  int brojUlaznih = 0;
  // obrađujemo jednu po jednu promenu
  int i = 0;
  while(i < promene.size()) {
    // obrađujemo sve intervale koji se završavaju, pa onda one
    // koji počinju u trenutnoj tački, ažurirajući broj ulaznih
    // itervala koji sadrže trenutnu tačku
    double xTrenutno = promene[i].first;
    while (i < promene.size() && promene[i].first == xTrenutno)
      brojUlaznih += promene[i++].second;

    // tekuća tačka jeste unutar nekog ulaznog intervala, ali
    // izlazni interval nije započet, pa ga zato započinjemo
    if (!zapocetIzlazni && brojUlaznih > 0) {
      zapocetIzlazni = true;
      xPocetakIzlaznog = xTrenutno;
    }
    // tekuća tačka nije unutar nijednog ulaznog intervala pa
    // se započeti izlazni interval mora da se završi na ovom mestu
    if (zapocetIzlazni && brojUlaznih == 0) {
      izlazni.push_back(make_pair(xPocetakIzlaznog, xTrenutno));
      zapocetIzlazni = false;
    }
  }
}

int main() {
  // učitavamo ulazne intervale
  int n;
  cin >> n;
  Intervali ulazni(n);
  for (int i = 0; i < n; i++) {
    double x1, x2;
    cin >> x1 >> x2;
    ulazni[i] = make_pair(x1, x2);
  }

  // ukrupnjavamo ulazne intervale
  Intervali izlazni;
  ukrupniIntervale(ulazni, izlazni);

  // izracunavamo ukupnu duzinu prekrivaca
  int duzina = 0;
  for (auto it : izlazni)
     duzina += it.second - it.first;
  cout << duzina << endl;
```

```
  // ispisujemo broj intervala u prekrivacu
  cout << izlazni.size() << endl;
  return 0;
}
```

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.

Naivan način da se zadatak reši je da se ispitaju sve trojke brojeva, što jasno dovodi do složenosti $O(n^3)$.

Ključna opaska u zadatku je da zbir minimuma svih trojki elemenata niza ne zavisi od redosleda elemenata u nizu. Naime, indeksi $0 \leq i < j < k < n$, i elementi na odgovarajućim pozicijama određuju zapravo sve tročlane (multi)podskupove skupa elemenata niza (dopušteno je ponavljanje elemenata). Permutovanjem niza, njegovih tročlani (multi)podskupovi se ne menjaju.

Određivanje traženog zbira najjednostavnije je ako je niz sortiran (na primer, u neopadajućem poretku). Naime, na osnovu prethodnog zbir elemenata originalnog i sortiranog niza je jednak. Nakon sortiranja, najmanji element niza a_0 biće manji od $n-1$ elemenata niza, pa će prema tome biti najmanji element u $(n-1) \cdot (n-2)/2$ trojki - toliko puta će ući u traženi zbir. Sledeći po veličini element a_1 će biti manji od $n-2$ elementa niza i stoga će biti najmanji u $(n-2) \cdot (n-3)/2$ trojki - toliko puta će ući u traženi zbir. Na kraju, a_{n-3} će biti najmanji u jednoj trojki - toliko puta će ući u traženi zbir. U opštem slučaju, element a_i biće manji od $n-i-1$ elemenata, pa će biti minimum $(n-i-1)(n-i-2)/2$ trojki. Prema tome, zbir minimuma svih trojki jednak je:

$$S = \sum_{i=0}^{n-3} a_i \cdot \frac{(n-i-1) \cdot (n-i-2)}{2}.$$

Na osnovu ovoga je jednostavno napraviti efikasnu implementaciju.

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

int zbirMinimumaTrojki(vector<int>& a) {
  // sortiramo niz
  sort(begin(a), end(a));

  // ukupan zbir minimuma trojki
```

```
  int zbir = 0;
  for (int i = 0; i < n - 2; i++) {
    // broj trojki u kojima je a[i] najmanji
    // medju n-1-i elemenata biramo sve parove
    int brojTrojki = brojParova(n - 1 - i);
    // uvećavamo zbir za sva pojavljivanja elementa a[i] kao
    // minimuma
    zbir += brojTrojki * a[i]
  }

  // vraćamo konačan rezultat
  return zbir;
}
```

Složenost ovog algoritma je prilično očigledno $O(n)$.

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 podrazumeva da se za svakog takmičara proanaliziraju svi drugi takmičari sve dok se ne nađe neki koji ima strogo više poena u obe discipline. Složenost najgoreg slučaja u ovom pristupu je jasno $O(n^2)$.

Bolje se rešenje može dobiti ako se niz pretprocesira tako što se na neki način sortira. Pretpostavimo da takmičare sortiramo na osnovu poena iz matematike (za početak pretpostavimo da svi imaju različit broj poena). Tada za svakog takmičara lako možemo da odredimo one koji od njega imaju više poena iz matematike (to su oni iza njega) kao i one koji imaju manje poena od njega iz matematike (to su oni ispred njega). Niko od takmičara levo od njega ne može da ugrozi njegov status odličnog takmičara. Što se tiče ovih desno od njega, neko od njih ga ugrožava samo ako ima strogo više poena iz programiranja iz njega. On će dakle, biti odličan ako i samo ako niko desno od njega nema strogo više poena iz programiranja tj. ako i samo ako je njegov broj poena veći ili jednak od broja poena svih takmičara desno od njega. Ako svaki put taj maksimum tražimo iznova, dobijamo ponovo algoritam kvadratne složenosti. Međutim, znamo da maksimum možemo računati inkrementalno. Naime, možemo ojačati induktivnu hipotezu i pretpostaviti da prilikom obrade tekućeg elementa znamo maksimum broja poena iz programiranja svih takmičara desno od njega. Tu induktivnu hipotezu možemo lako održati time što maksimum ažuriramo na osnovu na osnovu broja poena tekućeg takmičara.

Ostaje još pitanje šta raditi u slučaju kada nekoliko takmičara ima isti broj poena iz matematike. Pretpostavimo da takmičare obrađujemo u neopadajućem redosledu poena iz programiranja. Tada će tekući maksimalni broj poena iz

programiranja biti maksimalni broj poena svih studenata koji imaju veći ili jednak broj poena iz matematike i manji ili jednak broj poena iz programiranja od tekućeg. Tekući takmičar može imati strogo manje poena programiranja od tekućeg maksimuma, samo ako je taj maksimum ostvario neko ko ima strogo više poena iz matematike iz njega (jer oni koji imaju jednako poena kao on iz matematike imaju manje ili jednako poena iz programiranja) i taj takmičar ne može biti odličan. Sa druge strane, ako ima više ili jednako poena od tekućeg maksimuma, onda on jeste odličan, jer niko ko ima više strogo više poena iz programiranja od njega ne može imati strogo više poena i iz matematike (svi desno od njega imaju manje ili jednako poena iz programiranja od tekućeg maksimuma, a svi levo od njega imaju manji ili jednak broj poena iz matematike od njega).

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) {
  // sortiramo poene leksikografski,
  // nerastuće po matematici i neopadajuće po programiranju
  sort(begin(poeni), end(poeni),
       [](auto& p1, auto& p2) {
          return p1.mat > p2.mat ||
                 p1.mat == p2.mat && p1.prog < p2.prog;
       });

  // ukupan broj do sada pronađenih odličnih takmičara
  int brojOdlicnih = 0;
  // najveći do sada viđen broj poena iz programiranja
  int maxProg = 0;
  for (int i = 0; i < n; i++) {
    // niko od takmičara ispred tekućeg nije osvojio više poena
    // od maxProg iz programiranja (iako su u matematici možda
    // imali više poena od tekućeg takmičara)
    // ako je on osvoji bar toliko poena, onda ne postoji takmičar
    // koji od njega ima i strogo više poena iz matematike
    // i strogo više poena iz programiranja, pa je tekući takmičar
    // odličan
    if (poeni[i].prog >= maxProg) {
      // uvećavamo broj odličnih takmičara
      brojOdlicnih++;
```

```
      // ažuriramo maksimalni broj osvojenih poena iz programiranja
      maxProg = prog;
    }
  }

  // vraćamo ukupan broj odličnih takmičara
  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 (pošto se u nizu pozicije broje od nule, tražimo da element na poziciji h ima strogo više od h citata). Pošto je složenost faze sortiranja $O(n \log n)$, ovu proveru možemo uraditi i linearnom pretragom, ne narušavajući opštu složenost ovog algoritma (iako je binarna pretraga moguća).

Zaista, ako radove sortiramo u nerastući niz c po broju citata, h-indeks možemo izračunati kao najmanji broj h takav da je $c_h \leq h$. Ako je neki rad na poziciji h u sortiranom nizu i ako on ima c_h citata, znamo da postoji bar $h+1$ rad koji ima bar c_h citata (jer on ima bar c_h citata i ispred njega postoji tačno h radova koji imaju bar onoliko citata koliko i on, jer je niz nerastući). U nerastuće sortiranom nizu proveravamo sve indekse h od 0 do $n-1$, sve dok ne naiđemo na prvi za koji važi da je $c_h \leq h$. Kada se to desi, znamo da postoji tačno h radova koji imaju bar h citata, jer je za prethodnu poziciju važilo da je $c_{h-1} > h-1$, tj. $c_{h-1} \geq h$, pa je postojalo tačno h radova sa bar c_h citata, a to je bar h citata. Takođe, ne postoji $h+1$ rad sa bar $h+1$ citatom, jer svi radovi od pozicije h do kraja niza imaju strogo manje od $h+1$ citata, a ispred njih postoji samo h radova.

```
int hIndeks(const vector<int>& brojCitata) {
  // sortiramo radove na osnovu broju citata, nerastuće
  sort(brojCitata.begin(), brojCitata.end(), greater<int>());

  // Ako je h-indeks jednak h, tada h-ti po redu rad ima bar h
  // citata. Tražimo najveći broj koji zadovoljava taj uslov.
  int indeks = 0;
  while (indeks < n && brojCitata[indeks] > indeks)
    indeks++;

  // vraćamo rezultat
```

```
  return indeks;
}
```

Recimo i da ovaj problem ima rešenje u linearnoj složenosti.

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 c_h (koji ne mora biti sortiran) i da se odredi broj elemenata niza koji su veći ili jednaki h. Međutim, ovo bi dovelo do algoritma kvadratne složenosti, što želimo da izbegnemo. 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.

Dakle, odgovara nam da pretragu organizujemo unatrag i da za svaki broj h od n pa unazad do nule određujemo broj radova koji imaju bar h citata, zaustavljajući se kada pronađemo prvu vrednost h tako da je $B_h \geq h$. Ostaje pitanje kako izračunati vrednosti b_h i kako izračunati početnu vrednost B_n. To možemo uraditi u jednom prolazu kroz niz c (zapravo, niz c ne moramo ni pamtiti, već prilikom učitavanja njegovih članova možemo samo izračunavati elemente b_h za $0 \leq h < n$, i ujedno izračunati vrednost B_n). Ove vrednosti možemo pamtiti u nizu brojača. Niz ima $n + 1$ elemenat, pri čemu se na pozicijama od 0 do $n - 1$ izračunavaju vrednosti b_h, a na poziciji n se čuva vrednost B_n. Niz inicijalizujemo na nulu i svaki put kada učitamo broj citata nekog rada proveravamo da li je manji od n i ako jeste, uvećavamo brojač radova sa tim brojem citata (broj b_h), a u suprotnom uvećavamo broj radova sa bar n citata (broj B_n).

```
int hIndeks(const vector<int>& brojCitata) {
  // brojRadova[i] = broj radovakoji imaju tačno i citata
  // brojRadova[n] = broj radova koji imaju >= n citata
  vector<int> brojRadova(n + 1, 0);
  for (int i = 0; i < n; i++) {
    // učitavamo broj citata tekućeg rada
    int brojCitata;
    cin >> brojCitata;
    // ažuriramo statistiku o broju radova
    broj_radova[broj_citata < n ? broj_citata : n]++;
  }

  // pokušavamo da uspostavimo što veci h-indeks
  int indeks = n;
  // ukupnoRadova = broj radova koji imaju >= indeks citata
  int ukupnoRadova = brojRadova[n];
  while (ukupnoRadova < indeks) {
    // moramo da smanjimo h-indeks
    indeks--;
    // ažuriramo broj radova sa >= indeks citata
```

```
    ukupnoRadova += brojRadova[indeks];
  }

  // važi da je broj radova sa >= indeks citata >= indeks i
  // indeks je najveci broj za koji to važi, pa je on traženi
  // rezultat
  cout << h_indeks << endl;
}
```

Stabilnost sortiranja

Algoritam sortiranja se naziva *stabilnim* (engl. *stable*) ako se prilikom sortiranja zadržava originalni redosled elemenata sa istim ključem. Uobičajene implementacije sortiranja selekcijom (engl. selection sort), brzog sortiranja (engl. quick sort) i sortiranja pomoću hipa (engl. hip sort) nisu stabilne. Sa druge strane, uobičajene implementacije sortiranja umetanjem (engl. insertion sort) i sortiranja objedinjavanjem (engl. merge sort) jesu stabilne. Kod sortiranja umetanjem bitno je zaustaviti se čim tekući element stigne iza elementa koji ima isti ključ kao i on (time se održava relativni raspored dva elementa sa istim ključem). 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 (tako elementi iz leve polovine koji u originalnom nizu prethode elementima iz desne polovine sa istom vrednošću ključa nastavljaju da prethode tim elementima i u objedinjenom nizu).

Svaki se algoritam može napraviti stabilnim po cenu korišćenja dodatne memorije i malo dodatnog vremena. Naime, elementi niza se mogu proširiti njihovom pozicijom u nizu i funkcija poređenja elemenata se može proširiti tako da se u slučaju utvrđivanja jednakosti ključeva odnos odredi na osnovu pozicije u originalnom nizu (u tom proširenom poretku više nema međusobno jednakih elemenata).

Funkcija `sort` u jeziku C++ ne garantuje stabilnost (jer je obično zasnovana na kombinaciji algoritama QuickSort, HeapSort i InsertionSort). 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 $O(n \log n)$, a u suprotnom je $O(n(\log n)^2)$.

Stabilnost nam omogućava da sortiranje niza po više kriterijuma uradimo tako što ćemo prvo sortirati po jednom, pa onda po drugom kriterijumu.

Problem: Sortirati niz intervala tako da budu sortirani po levom kraju, a ako im je levi kraj isti, onda po desnom kraju.

Jedan način je da se upotrebi funkcija poređenja koja vrši leksikografsko poređenje. Pretpostavićemo da su intervali zadati strukturom u kojoj se čuva početak i kraj.

```
struct Interval {
   int pocetak, kraj;
};

void sortirajIntervale(vector<Interval>& intervali) {
   sort(begin(intervali), end(intervali),
       [](auto &a, auto &b) {
           return a.pocetak < b.pocetak ||
                  a.pocetak == b.pocetak && a.kraj < b.kraj;
       });
}
```

Ako bi se umesto struktura upotrebili parovi, ne bi čak bilo neophodno ni navoditi funkciju poređenja (jer se parovi podrazumevamo porede leksikogrfski).

Drugi način je da se prvo sortira na osnovu desnog kraja intervala, a onda da se stabilnim sortiranjem sortira na osnovu levog kraja.

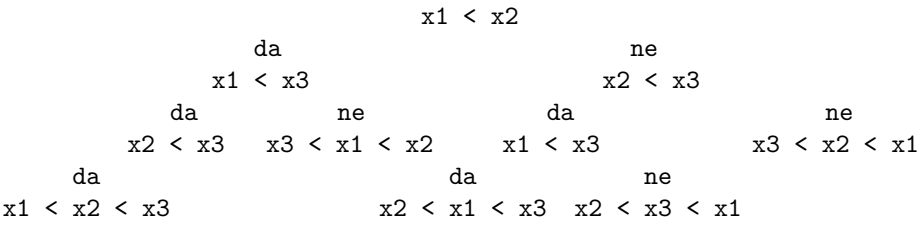
```
void sortirajIntervale(vector<Interval>>& intervali) {
   sort(begin(intervali), end(intervali),
       [](auto &a, auto &b) {
           return a.kraj < b.kraj;
       });
   stable_sort(begin(intervali), end(intervali),
       [](auto &a, auto &b) {
           return a.pocetak < b.pocetak;
       });
}
```

U ovom konkretnom primeru, bolja je svakako prva varijanta u kojoj se sortiranje vrši samo jednom, ali kako ćemo uskoro videti, u svetlu postojanja stabilnih algoritama sortiranja koji pod pretpostavkom da se koristi dodatna memorija i da je skup ključeva specifičan mogu biti i linearne složenosti, postizanje leksikografskog poretka ovakvim višekratnim stabilnim sortiranjem može biti efikasnije od klasičnog jedonkratnog sortiranja.

Donja granica složenosti sortiranja

Algoritmi sortiranja koje smo do sada sreli su imali vremensku složenost $O(n^2)$ ili $O(n \log n)$. Postavlja se pitanje da li je moguće napraviti algoritam koji bi imao manju složenost (na primer, $O(n)$), a odgovor na to pitanje zavisi od mnogih pitanja (na primer, da li nam je dopušteno da koristimo dodatnu memoriju, da li vrednosti niza dolaze iz nekog relativnog malog skupa i slično). 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 $\Theta(n \log n)$.

Preciznije, 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.



Veoma jednostavno je moguće dokazati da određivanje rasporeda n elemenata drvetom odlučivanja, tj. samo upoređivanjem parova elemenata niza zahteva bar $\Theta(n \log n)$ upoređivanja. Naime, broj upoređivanja u najgorem slučaju odgovara visini drveta. Pošto za niz od n elemenata postoji $n!$ mogućih rasporeda, drvo odlučivanja mora da ima $n!$ listova. Pošto drvo visine k ima najviše 2^{k-1} listova, visina drveta koje ima $n!$ listova je bar $\log n!$, što je jednako $\log 1 + \log 2 + ... + \log n > \log n/2 + ... + \log n > \log n/2 + ... + \log n/2 = (n/2) \log n/2 = \Theta(n \log n)$.

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 (u suprotnom bismo mogli upotrebiti mapu zasnovanu na balansiranim drvetima ili heš-tablici).

```
void sortiranjePrebrojavanjem(vector<int>& a) {
  int m = *max_element(begin(a), end(a));
  // brojimo pojavljivanja svakog elementa niza
  vector<int> frekvencije(m+1, 0);
  for (int i = 0; i < n; i++)
    frekvencije[a[i]]++;

  // rekonstruisemo niz iz pocetka
  int k = 0;
```

```
  for (int j = 0; j < m; j++)
    for (int i = 0; i < frekvencije[j]; i++)
      a[k++] = j;
}
```

Složenost faze prebrojavanja je $O(n)$, a složenost rekonstrukcije je $O(m)$, pa je ukupna složenost jednaka $O(n + m)$. Ako m nije za red veličine veće u odnosu na n, ovaj algoritam je praktično linearne složenosti, što je bolje nego klasični algoritmi sortiranja.

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. Dakle opet je centralni korak u algoritmu sortiranja prebrojavanje broja elemenata niza koji odgovaraju svakom mogućem ključu. Ovakav postupak sortiranja nazivaćemo *sortiranje razvrstavanjem*, mada se na engleskom jeziku ono naziva *sortiranje prebrojavanjem* (engl. *counting sort*). Prikažimo ideju sortiranja pisama razvrstavanjem na jednom veoma sličnom zadatku.

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.

Osnovno pitanje je kako čuvati broj birača za svaku opštinu. Pošto ne znamo unapred spisak opština, najbolje rešenje je da koristimo uređenu mapu. Time ujedno opštine možemo obilaziti u leksikografskom redosledu naziva. Jednom kada prebrojimo birače na svakoj opštini, prolazimo redom kroz niz opština i za svaku opštinu početnu poziciju elemenata u nizu dobijamo održavajući tekući zbir do sada obrađenih opština. Pozicije beležimo u novoj mapi. Nakon toga razvrstavamo elemente originalnog niza birača i svakog birača upisujemo na tekuću poziciju pridruženu njegovoj opštini i tu poziciju uvećavamo (kako bi naredni birač sa te opštine bio upisan na naredno mesto u nizu).

```
// za svakog birača poznata je opština sa koje dolazi i šifra
struct Birac {
```

```
  string opstina;
  string sifra;
};

void sortiraj(vector<Birac>& biraci) {
  // izračunavamo broj birača iz svake opštine
  // preslikvanje brojBiraca određuje broj birača za dati
  // naziv opštine
  map<string, int> brojBiraca;
  // prolazimo kroz spisak svih birača
  for (auto birac : biraci)
    // uvecavamo broj birača na opštini trenutnog birača
    brojBiraca[birac.opstina]++;

  // izračunavamo poziciju u sortiranom nizu na kojoj
  // počinju birači sa date opštine
  map<string, int> pozicije;
  // ukupan broj birača u do sada obrađenim opštinama
  int prethodnoBiraca = 0;
  // prolazimo kroz sve opštine u sortiranom redosledu
  // (abecedno, leksikografski)
  for (auto it : brojTakmicara) {
    // tekuća opština počinje na poziciji određenoj
    // brojem takmičara u prethodnim opštinama
    pozicije[it.first] = prethodnoBiraca;
    // uvećavamo broj birača u prethodnim opštinama za broj
    // birača u trenutnoj opštini, pripremajući se za novu
    // iteraciju
    prethodnoBiraca += it.second;
  }

  // konačan sortirani niz birača
  vector<Birac> sortirano(biraci.size());
  // prolazimo kroz sve birače iz polaznog niza
  for (auto birac : biraci) {
    // postavljamo birača na tekuće mesto u njegovoj opštini
    sortirano[pozicije[birac.opstina]] = birac;
    // uvecavamo slobodnu poziciju u toj opštini
    pozicije[birac.opstina]++;
  }

  biraci = sortirano;
}
```

Na sličan način možemo rešiti i naredno uopštenje problema trobojke, o kom smo ranije diskutovali.

Problem: Date tačke $t_0, t_1, \ldots, t_{k-1}$ dele realnu pravu na intervale $(-\infty, t_0)$, $[t_0, t_1)$, ..., $[t_{k-2}, t_{k-1})$, $[t_{k-1}, +\infty)$. Napiši program koji sortira unete brojeve tako da se prvo pojavljuju brojevi iz prvog intervala, zatim brojevi iz drugog

i tako dalje, pri čemu je sortiranje stabilno, tj. u svakoj grupi se pojavljuju brojevi u istom redosledu kako su uneti.

Ideja nam je da elemente originalnog niza prekopiramo u drugi, pomoćni niz u traženom redosledu. Za to je potrebno da za svaki od $k+1$ zadatih intervala znamo na kojoj poziciji u rezultujućem nizu treba da budu smešteni elementi tog intervala. Na početku ćemo za svaki interval odrediti koliko elemenata originalnog niza pripada tom intervalu. Na osnovu toga, lako možemo odrediti početnu poziciju elemenata svakog intervala u rezultujućem nizu (u pitanju su parcijalni zbirovi niza frekvencija). Nakon toga prolazimo kroz elemente originalnog niza, određujemo kom intervalu pripadaju, postavljamo ga u rezultujući niz, na poziciju koja je trenutno izračunata kao početna za elemente tog intervala i nakon toga tu početnu poziciju uvećavamo za 1 (da bi naredni element tog intervala, bio upisan iza ovoga).

Interval kome pripada svaki od elemenata možemo efikasno odrediti binarnom pretragom.

```
// redni broj intervala
// (-inf, t_0), [t0, t1), ..., [t_{k-1}, +inf)
// kojem pripada tačka x
int interval(const vector<int>& t, int x) {
  return distance(t.begin(), upper_bound(t.begin(), t.end(), x));
}

// t - tačke podele t0, ..., t{k-1} koje dele pravu na intervale
// (-inf, t_0), [t0, t1), ..., [t_{k-1}, +inf)
// a - elementi koje treba razvrstati
vector<int> nBojka(vector<int>& t, vector<int>& a) {
  // dužina nizova
  int k = t.size(), n = a.size();

  // za svaki interval podele (ima ih k+1) određujemo broj elemenata
  // niza koji mu pripadaju
  vector<int> frekvencije(k+1, 0);
  for (int i = 0; i < n; i++)
    frekvencije[interval(t, a[i])]++;

  // određujemo početne pozicije elemenata svakog intervala
  vector<int> pozicije(k+1);
  pozicije[0] = 0;
  for (int i = 0; i < frekvencije.size(); i++)
    pozicije[i+1] = pozicije[i] + frekvencije[i];

  // kopiramo niz a u niz b, na odgovarajuće pozicije
  vector<int> b(n);
  for (int i = 0; i < n; i++)
    b[pozicije[interval(t, a[i])]++] = a[i];
}
```

Prodiskutujmo složenost prethodnog algoritma. Brojanje elemenata svakog intervala zahteva memoriju $O(k)$ i vreme $O(n \log k)$. Izračunavanje pozicija na osnovu frekvencija zahteva $O(k)$ vremena i dodatnih $O(k)$ memorije, dok kopiranje u rezultujući niz zahteva $O(n)$ dodatne memorije i $O(n \log k)$ vremena. Ukupna memorijska složenost je, dakle, $O(n + k)$, a vremenska $O(n \log k + k)$.

Recimo i da nam nakon izračunavanja početnih pozicija više nije neophodan vektor frekvencija, pa je za smeštanje i jednih i drugih moguće upotrebiti isti niz. Ako se izbegne smeštanje početne nule u nizu pozicija, onda svaki element niza pozicija predstavlja poziciju u rezultujućem nizu neposredno iza pozicije na koju treba smestiti poslednji element iz odgovarajuće grupe. U poslednjoj petlji elemente originalnog niza obilazimo unatrag i pre upisivanja tekućeg elementa umanjimo poziciju na koju se postavlja.

```
vector<int> nBojka(vector<int>& t, vector<int>& a) {
  // dužina nizova
  int k = t.size(), n = a.size();

  // za svaki interval podele (ima ih k+1) određujemo broj
  // elemenata niza koji mu pripadaju
  vector<int> broj(k+1, 0);
  for (int i = 0; i < n; i++)
    broj[interval(t, a[i])]++;

  // određujemo pozicije neposredno iza bloka elemenata
  // iz svakog intervala
  for (int i = 1; i < frekvencije.size(); i++)
    broj[i+1] += broj[i];

  // kopiramo niz a u niz b, na odgovarajuće pozicije
  vector<int> b(n);
  for (int i = n-1; i >= 0; i--)
    b[--pozicije[interval(t, a[i])]] = a[i];
}
```

Sortiranje višestrukim razvrstavanjem

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 26^2 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. Razmotrimo kako bi to izgledalo na sledećem primeru.

```
Ana Marković
Lazar Marković
Ilija Bojković
Cile Petrović
Aljoša Milutinović
Luna Backović
Andrej Jovanović
Jovana Vojvodić
Katarina Pavlović
```

U prvoj fazi sortiranje bismo izvršili na osnovu prezimena.

```
Ilija Bojković
Luna Backović
Andrej Jovanović
Ana Marković
Lazar Marković
Aljoša Milutinović
Cile Petrović
Katarina Pavlović
Jovana Vojvodić
```

Nakon toga sortiranje vršimo na osnovu imena.

```
Andrej Jovanović
Ana Marković
Aljoša Milutinović
Cile Petrović
Ilija Bojković
Jovana Vojvodić
Katarina Pavlović
Luna Backović
Lazar Marković
```

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.

Algoritam koji sortiranje vrši tako što se sortiranje prebrojavanjem primenjuje uzastopno, po raznim ključevima nazivaćemo *sortiranje višestrukim razvrstavanjem*. Na engleskom jeziku algoritam se naziva radix sort, zato što se najčešće primenjuje na cifre u zapisu broja (kao što ćemo videti u narednom zadatku).

```
// Ucenik je tip predstavljen dvočlanim nizom
// na poziciji IME nalaziće se ime, a na poziciji PREZIME prezime
enum {IME = 0, PREZIME = 1};
typedef array<string, 2> Ucenik;

// Razvrstava niz učenika i to na osnovu inicijala i to na osnovu:
// prvog slova imena ako je ip = 0 tj. prezimena ako je ip = 1.
// U okviru grupe učenika sa istim inicijalom originalni redosled se
// održava.
vector<Ucenik> razvrstaj(const vector<Ucenik>& ucenici, int ip) {
  // izračunavamo broj pojavljivanja imena i prezimena koje
  // počinju na svako od 26 slova
  int brojPojavljivanja[26] = {0};
  for (auto ucenik : ucenici)
    brojPojavljivanja[ucenik[ip][0] - 'a']++;

  // izračunavamo pozicije na kojima u rezultujućem nizu počinju
  // da seređaju reči koje počinju na svako od 26 slova
  int pozicija[26];
  pozicija[0] = 0;
  for (int i = 1; i < 26; i++)
    pozicija[i] = pozicija[i-1] + brojPojavljivanja[i-1];

  // rezultujući vektor
  vector<Ucenik> rezultat(ucenici.size());

  // prepisujemo sve reči iz vektora ucenici u vektor rezultat, na
  // odgovarajuće pozicije
  for (auto ucenik : ucenici)
    rezultat[pozicija[ucenik[ip][0] - 'a']++] = ucenik;

  // vraćamo konačan rezultat
  return rezultat;
}

int main() {
  ...
  // razvrstavamo po prezimenu
  ucenici = razvrstaj(ucenici, PREZIME);
  // razvrstavamo po imenu
  ucenici = razvrstaj(ucenici, IME);
  ...
}
```

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

Prikažimo kako se radiks sort može primeniti na sledeći niz brojeva.

37 140 7 10 99 102 17 25 1 48 14 3 18

Prvo vršimo sortiranje na osnovu cifre jedinica

140 10 1 102 3 14 25 37 7 17 48 18 99

Nakon toga, na osnovu cifre desetica.

1 102 3 7 10 14 17 18 25 37 140 48 99

Na kraju, sortiranje vršimo na osnovu cifre stotina.

1 3 7 10 14 17 18 25 37 48 99 102 140

Algoritam možemo implementirati na sledeći način.

```
void sortiranjeRazvrstavanjem(vector<int>& a, int s) {
  // brojimo pojavljivanje svake cifre uz koeficijent s
  // (on je uvek stepen desetke)
  int broj[10] = {0};
  for (int i = 0; i < a.size(); i++)
    broj[(a[i] / s) % 10]++;
  // pozicije u rezultujućem nizu ispred kojih se završavaju grupe
  // elemenata sa odgovarajućim ciframa uz koeficijent s
  for (int i = 1; i < 10; i++)
    broj[i] += broj[i-1];
  // prepisujemo elemente u pomoćni niz
  vector<int> pom(a.size());
  for (int i = a.size() - 1; i >= 0; i--)
    pom[--broj[(a[i] / s) % 10]] = a[i];
  // vraćamo elemente nazad u originalni niz
  a = pom;
}

void sortiranjeVisestrukimRazvrstavanjem(vector<int>& a) {
  // određujemo maksimum niza a
  int max = numeric_limits<int>::min();
  for (int x : a)
    if (x > max)
      max = x;

  // sortiramo na osnovu svake cifre krenuvši od cifara jedinica
  for (int s = 1; max / s > 0; s *= 10)
    sortiranjeRazvrstavanjem(a, s);
}
```

Primene binarne pretrage

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 $a_i = i$ (pozicije se broje od nule). Ako pozicija postoji vratiti je, a ako ne postoji, vratiti -1.

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 $a_{i+1} \geq a_i + 1$. Slično je $a_{i+2} > a_{i+1}$, pa je $a_{i+2} \geq a_{i+1} + 1 \geq a_i + 2$. Nastavljanjem ovog rezona može se zaključiti da je $a_j \geq a_i + j - i$, odnosno $a_j - j \geq a_i - i$. Rešenje, dakle, možemo odrediti tako što binarnom pretragom proverimo da li niz $a_i - i$ sadrži nulu i ako sadrži, tada je rešenje pozicija na kojoj se nalazi nula.

Binarnu pretragu možemo realizovati bilo bibliotečkim funkcijama, bilo ručnom implementacijom. U slučaju bibliotečke funkcije najlakše nam je da niz transformišemo tako što od svakog elementa oduzmemo njegov indeks i onda upotrebimo funkciju `lower_bound` da pronađemo prvu poziciju koja je nenegativna i onda proverimo da li je ona unutar niza i da li je jednaka nuli.

```
int itiNaMestuI(vector<int>& a) {
  // pripremamo niz ga za pretragu a[k] = k akko je a[k] - k = 0
  // tako da umesto niza a, pretražujemo neopadajući niz a[i] - i
  for (int i = 0; i < n; i++)
    a[i] -= i;

  // tražimo poziciju nule u transformisanom nizu tako što
  // pronalazimo poziciju prvog elementa koji je >= 0
  auto it = lower_bound(a.begin(), a.end(), 0);

  // ako takav element postoji i ako je jednak nuli
  if (it != a.end() && *it == 0)
    // pronašli smo element i izračunavamo njegovo rastojanje
    // od početka niza
    return distance(a.begin(), it);
  else
    // u suprotnom element ne postoji u nizu
    return -1;
}
```

Kod ručne implementacije nema potrebe da modifikujemo niz, jer nakon nalaženja središnjeg elementa možemo proveriti da li je $a_s < s$ (tada bi u transformisanom nizu važilo da je $a_s - s < 0$, pa je pretragu potrebno nastaviti desno od pozicije s) ili je $a_s \geq s$ (tada bi u transformisanom nizu važilo da je $a_s - s \geq 0$, pa je pretragu potrebno nastaviti levo od pozicije s).

```
int itiNaMestuI(const vector<int>& a) {
  // sprovodimo binarnu pretragu - ako traženi element a[i] = i
  // postoji, on se nalazi u intervalu [l, d]
  // invarijanta:
  //  za sve elemente u intervalu [0, l) važi da je a[i] < i
  //  za sve elemente u intervalu (d, n) važi da je a[i] >= i
  int l = 0, d = n-1;
  // dok interval [l, d] nije prazan
  while (l <= d) {
    // pronalazimo sredinu intervala
    int s = l + (d - l) / 2;
    if (a[s] < s)
      // najmanji element takav da je a[i]=i
      // može biti samo desno od s
      l = s + 1;
    else
      // najmanji element takav da je a[i]=i
      // može biti samo levo od s
      d = s - 1;
  }
  // svi elementi levo od l su takvi da je a[l] < l
  // ako postoji element takav da je a[l] = l,
  // najmanji takav može biti jedino na poziciji i
  if (l < n && a[l] == l)
    return l;
  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 kvadrat. Složenost tog pristupa bi bila $O(n^4)$.

Bolji način je da se za svaki par tačaka proveri da li čini dijagonalu kvadrata sastavljenog od tačaka iz tog skupa. Ako dva naspramna temena kvadrata imaju koordinate (x_1, y_1) i (x_2, y_2), vektor koji ih spaja ima koordinate (d_x, d_y) gde je $d_x = x_2 - x_1$ i $d_y = y_2 - y_1$. Do dva preostala temena kvadrata može stići tako što se početno teme prvo translira do njihovog središta, translacijom za vektor $(d_x/2, d_y/2)$, a zatim se translira za vektor koji je rotacija tog vektora za 90 stepeni (ako rotiramo u jednom smeru dobijamo jedno, a ako rotiramo u drugom smeru dobijamo drugo teme). Te dve rotacije su $(-d_y/2, d_x/2)$ i $(d_y/2, -d_x/2)$.

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 (na primer, u leksikografskom redosledu koordinata, tj. po koordinati x tj. y ako su x koordinate jednake) i da se primeni binarna pretraga.

Pošto će svaki kvadrat biti pronađen dva puta (po jednom za svaku svoju dijagonalu) 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. Naivno rešenje bi svaki put prolazilo kroz ceo niz poena i brojalo one takmičare čiji su poeni veći ili jednaki trenutnom pragu, čime bi se dobio algoritam ukupne složenosti $O(mn)$. Ako se niz poena sortira nakon učitavanja, brojanje takmičara iznad praga se može ostvariti mnogo efikasnije. Kada se pronađe pozicija prvog takmičara čiji su poeni iznad praga, znamo da su svi takmičari od te pozicije pa do kraja niza primljeni (jer je niz sortiran, pa svi imaju više ili jednako poena od broja poena na toj poziciji) i da niko ispred te pozicije nije primljen (jer je to prva pozicija na kojoj je broj poena iznad praga). Prvu vrednost veću ili jednaku datoj u sortiranom nizu možemo jednostavno odrediti binarnom pretragom, pa je složenost ovog pristupa jednaka $O((n+m)\log n)$, što je dosta efikasnije u slučaju kada je m veliki broj (a lošije je od naivnog rešenja u slučaju kada je m mali broj).

```
// učitavamo poene u niz
int n;
cin >> n;
vector<int> poeni(n);
for (int i = 0; i < n; i++)
  cin >> poeni[i];

// sortiramo poene
sort(begin(poeni), end(poeni));

// učitavamo jedan po jedan prag
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int prag;
  cin >> prag;
  // pronalazimo prvi element u nizu poena koji je veći ili
  // jednak pragu i računamo njegovo rastojanje do kraja niza
  auto it = lower_bound(begin(poeni), end(poeni), prag);
  cout << distance(it, end(poeni)) << endl;
}
```

Kružne zone

Problem: Kvalitet signala zavisi od udaljenosti tačke od predajnika. 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 (zone se broje od nule, ako je tačka na granici dve zone smatra se da pripada unutrašnjoj, a ako ne pripada ni jednoj zoni ispisuje se -1).

Tačka pripada nekoj zoni ako i samo ako je njeno rastojanje od koordinatnog početka (a njega možemo izračunati korišćenjem Pitagorine teoreme) strogo veće od unutrašnjeg, a manje ili jednako spoljašnjem prečniku te zone. Unutrašnji prečnik neke zone jednak je zbiru širina svih zona pre nje, ne uključujući njenu širinu, a spoljašnji prečnik jednak je zbiru širina svih zona pre nje, uključujući i njenu širinu. Primetimo da je unutrašnji prečnik neke zone jednak spoljašnjem prečniku prethodne zone (osim kod zone 0, koja ima unurašnji prečnik nula). Zato je dovoljno da izračunamo spoljašnje prečnike svih zona, a oni se mogu izračunati kao parcijalni zbirovi širina zona. Njih možemo računati inkrementalno, dodajući svaku novu učitanu širinu na tekući zbir ili, u jeziku C++ 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, bilo ručno implementiranom bilo pomoću bibliotečke funkcije `lower_bound`.

```
void indeksiZona(const vector<double>& zone,
                 const vector<pair<double, double>>& tacke,
                 vector<int>& indeksi) {
  // izračunavamo poluprečnike zona
  // (kao zbir širina svih zona zaključno sa tom)
  vector<double> poluprecnici(zone.size());
  partial_sum(zone.begin(), zone.end(), poluprecnici.begin());

  // obrađujemo jednu po jednu tačku
  for (int i = 0; i < tacke.size(); i++) {
    int x = tacke[i].first, y = tacke[i].second;
    // rastojanje tačke (x, y) od koordinatnog pocetka
    double r = sqrt(x*x + y*y);

    // pronalazimo prvu zonu takvu da joj je poluprečnik veći
    // ili jednak r
    auto it = lower_bound(poluprecnici.begin(), poluprecnici.end(), r);

    // ako takva ne postoji, tacka je izvan svih zona
    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).

Najbliži datom broju može biti ili prvi veći ili poslednji manji od broja. Bilo

koji od njih možemo naći binarnom pretragom i zatim proveriti onaj drugi (ako postoji).

U narednoj implementaciji prvi element koji je veći ili jednak od x tražimo bibliotečkom funkcijom `lower_bound`.

```
int najbliziDatom(const vector<int>& a, int x) {
  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);
}
```

Složenost ovog algoritma potiče od binarne pretrage i iznosi $O(\log n)$.

Varijacije binarne pretrage

Najmanji element rotiranog sortiranog niza

Problem: Sortirani niz u kome su svi elementi različiti je rotiran za k mesta ulevo i time je dobijen ciklični niz koji zadovoljava uslov da je $x_k < x_{k+1} < ... < x_n < x_0 < ... < x_{k-1}$. Naći njegov najmanji element.

```
11 13 15 19 24 1 3 8 9
```

Zadatak možemo rešiti u složenosti $O(n)$ klasičnim algoritmom za pronalaženja minimuma (ili bibliotečkom funkcijom `min_element`). Bolje rešenje se može dobiti binarnom pretragom. Nakon rotacije svi elementi u početnom delu niza su strogo veći od početnog, a onda u završnom delu niza idu svi elementi koji su strogo manji od početnog. Najmanji element koji tražimo je prvi element u nizu koji je strogo manji od početnog i njega možemo naći binarnom pretragom u složenosti $O(\log n)$. Treba obratiti pažnju na specijalni slučaj u kome je niz rotiran za 0 mesta i tada ne postoji element koji je strogo manji od početnog. Binarna pretraga će tada vratiti poziciju iza kraja niza i u tom slučaju najmanji element u nizu je upravo prvi element niza.

```
// pronalazi najmanji element u rotiranom sortiranom nizu
int najmanji(const vector<int>& a) {
  int n = a.size();
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d-l)/2;
```

```
    if (a[s] < a[0])
      d = s-1;
    else
      l = s+1;
  }
  return l < n ? a[l] : a[0];
}
```

Invarijanta petlje je da su svi elementi ispred pozicije l veći ili jednaki početnom i sortirani su, dok su svi elementi iza pozicije d strogo manji od početnog i sortirani su. Pretraga se završava u trenutku kada je $l = d + 1$. Svi elementi iza pozicije d tj. svi elementi od pozicije l pa do kraja niza strogo manji od početnog elementa niza, dok su elementi od početka niza levo od pozicije l veći ili jednaki od početnog elementa niza. Ako postoje elementi od pozicije l do kraja niza, tj. ako je $l < n$, tada su oni sigurno manji od elemenata ispred pozicije l, a pošto su sortirani, najmanji je prvi od njih, tj. element na poziciji l. U suprotnom, ako je $l = n$, tada postoji samo levi deo niza, tj. svi elementi u nizu su veći ili jednaki početnom elementu, a pošto je taj deo niza sortiran, najmanji element je početni.

Provera ovog specijalnog slučaja se može izbeći ako se umesto odnosa sa levim, gleda odnos sa desnim krajem niza.

```
// pronalazi najmanji element u rotiranom sortiranom nizu
int najmanji(const vector<int>& a) {
  int n = a.size();
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d-l)/2;
    if (a[s] < a[n-1])
      d = s-1;
    else
      l = s+1;
  }
  return a[l];
}
```

Invarijanta je da su svi elementi ispred pozicije l veći od poslednjeg elementa niza i sortirani su, dok su svi elemeni iza pozicije d manji ili jednaki od poslednjeg elementa niza i sortirani su. Kada se petlja završi važi da je $l = d + 1$. Zato su svi elementi iza pozicije l strogo veći od elemenata na poziciji l i onima iza nje. Pošto je deo od pozicije l do kraja sortiran, minimum se nalazi na poziciji l, jer je taj deo uvek neprazan. Zaista, mora da važi da je $l < n$, jer bi u suprotnom poslednji deo bio levo od pozicije l, što je nemoguće, jer su u tom delu nalaze elementi koji su strogo veći od poslednjeg.

Pretraga rotiranog sortiranog niza

Problem: Sortirani niz u kome su svi elementi različiti je rotiran za proizvoljni broj mesta. Proveriti da li u njemu postoji dati element.

Naivan način da se zadatak reši je da se zanemari sortiranost i da se primeni linearna pretraga, međutim to je neefikasno (jer je složenost $O(n)$). Bolje rešenje je da se primeni binarna pretraga. Na osnovu prethodnog zadatka moguće je binarnom pretragom u složenosti $O(\log n)$ naći poziciju najmanjeg elementa u nizu. Time je niz podeljen na dva strogo rastuća segmenta i na svaki od njih je moguće zasebno primeniti klasičnu binarnu pretragu za traženim elementom. Vrše se tri binarne pretrage, pa je složenost $O(\log n)$.

Moguće je napraviti i rešenje koje koristi samo jednu binarnu pretragu. Kao i uvek kod binarne pretrage, cilj nam je da jednim pitanjem eliminišemo bar pola elemenata niza. Pretpostavimo da pretražujemo segment određen pozicijama $[l, d]$ i neka je s središnja pozicija u tom segmentu. Ako se traženi element nalazi na poziciji s, pretraga je uspešna. U suprotnom su nam ostala dva segmenta niza (segment određen pozicijama $[l, s-1]$ i segment određen pozicijama $[s+1, d]$). Ključni uvid je da je bar jedan od ta dva dela niza sortiran (a mogu biti sortirana i oba). Prvi segment je sortiran ako je $a_l < a_{s-1}$, a drugi ako je $a_{s+1} < a_d$. Ako je sortiran segment određen pozicijama $[l, s-1]$, tada proveravamo da li element može biti u njemu (to je slučaj kada je $a_l \leq x \leq a_{s-1}$). Ako može, onda smo sigurni da da element nije u drugom delu niza (jer znamo da su svi elementi u segmentu određenom pozicijama $[s+1, d]$ ili strogo manji od a_l ili strogo veći od a_{s-1}) i pretragu svodimo na segment $[l, s-1]$. Ako x nije u tim granicama, onda nije u tom segmentu, pa pretragu svodimo na segment određen pozicijama $[s+1, d]$. Situaciju u kojoj je sortiran segment $[s+1, d]$ obrađujemo potpuno analogno.

```
bool sadrzi(const vector<int>& a, int x) {
  int n = a.size();
  int l = 0, d = n-1;
  while (l <= d) {
    // središte segmenta [l, d]
    int s = l + (d-l)/2;
    if (a[s] == x)
      // element je pronađen na poziciji s
      return true;
    if (a[l] < a[s-1]) {
      // sortiran je segment [l, s-1]
      if (a[l] <= x && x <= a[s-1])
        // x pripada rasponu elemenata segmenta [l, s-1]
        // pa pretragu svodimo na taj segment
        d = s-1;
      else
        // x ne pripada segmentu [l, s-1]
        // pa jedino može pripasti segmentu [s+1, d]
```

```
        l = s+1;
    } else {
      // sortiran je segment [s+1, d]
      if (a[s+1] <= x && x <= a[d])
        // x pripada rasponu elemenata segmenta [s+1, d]
        // pa pretragu svodimo na taj segment
        l = s+1;
      else
        // x ne pripada segmentu [s+1, d]
        // pa jedino može pripasti segmentu [l, s-1]
        d = s-1;
    }
  }
  return false;
}
```

Naglasimo da je pretpostavka da su elementi različiti (u suprotnom ne možemo konstruisati rešenje logaritamske složenosti).

Brojevi najbliži datom

Problem: Dat je sortirani niz od n brojeva. Odrediti poziciju na kojoj počinje k brojeva najbližih datom (ako postoji više takvih pozicija, ispisati najmanju).

Moglo bi se pomisliti da je ovo jednostavno uopštenje prethodnog zadatka i da se problem može rešiti tako što se nađe pozicija elementa najbližeg datom i onda se od njega proširimo u jednom ili drugom smeru. Međutim, ovde treba biti obazriv. Naime ta faza proširivanja složenosti je $O(k)$ i ako je k veliko (reda veličine n) dovodi do neefikasnog algoritma linearne složenosti.

Mnogo bolje rešenje je zasnovano na sledećoj važnoj činjenici. Do jednog trenutka u nizu se isplati pomerati početak segmenta od k elemenata nadesno, a onda se od jednog trenutka to više ne isplati. Početak traženog segmenta će onda biti prva pozicija na kojoj se više ne isplati pomerati početak nadesno. Glavno pitanje je kako za dati početak niza dati odgovor na pitanje da li se isplati pomeriti ga nadesno. Neka k elemenata niza počinju na poziciji i tj. neka su to elementi $a_i, \ldots, a_{i+k-1}$. Pomeranje nadesno je moguće samo ako je $i+k < n$ i njime se element a_i izbacuje iz segmenta i umesto njega se u segment ubacuje element a_{i+k}. To se isplati ako je a_{i+k} bliži elementu x od elementa a_i tj. ako je $|a_i - x| > |a_{i+k} - x|$. Ako je razdaljina jednaka, biramo da segment ne pomeramo, jer se u zadatku traži najmanja pozicija.

Na primer, ako je dat niz $1, 3, 7, 9, 13, 22, 27, 28$ i tražimo 4 elementa koji su najbliži elementu 15, pomeranje sa prve pozicije podrazumeva izbacivanje elementa 1 i dodavanje elementa 13, što se svakako isplati jer razlika pada sa 14 na 2. Pomeranje na sledeću poziciju se takođe isplati, jer izbacivanjem elementa 3 i dodavanje elementa 22 razlika sa 12 pada na 7. Pomeranje na narednu poziciju se ne isplati, jer izbacivanjem elementa 7 i dodavanjem elementa 27 razlika se

sa 8 penje na 12. Ni dalja pomeranja se ne isplate. Naime, naredno pomeranje bi izbacilo element 13 i ubacilo element 28, pa bi se razlika sa 2 popela na 13. Ako bismo tražili elemente najbliže vrednosti 0, već prvo pomeranje se ne bi isplatilo, jer bi se izbacivanjem elementa 1 i ubacivanjem elementa 13 razlika sa 1 popela na 13. Slično, ako bi se tražila 4 elementa najbliža vrednosti 30, svako pomeranje nadesno bi se isplatilo. Na primer, poslednje pomeranje bi izbacilo element 9 i ubacilo element 28 čime bi se razlika sa 21 smanjila na 2.

Jedan način da nađemo prvu poziciju za koju se pomeranje ne isplati je da primenimo linearu pretragu (kao u primeru) i krenuvši od prve, analiziramo jednu po jednu narednu poziciju, sve dok se pomeranje isplati. Pošto imamo situaciju u kojoj prvo idu elementi koji zadovoljavaju neki uslov, a zatim elementi koji ne zadovoljavaju taj uslov, moguće je primeniti i binarnu pretragu za prelomnom tačkom tj. binarnu pretragu za prvim elementom koji ne zadovoljava uslov pomeranja.

```
int k_najblizih_datom(const vector<int>& a, int x, int k) {
  int l = 0, d = a.size() - k - 1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (abs(a[s] - x) > abs(a[s+k] - x))
      l = s + 1;
    else
      d = s - 1;
  }
  return l;
}
```

Rang svakog elementa

Problem: Nakon takmičenja iz informatike poznata je lista učenika sa poenima (svaki učenik ima različit broj poena), međutim, nije poznat plasman (redni broj svakog učenika). Napiši program koji za svakog učenika određuje plasman i ispisuje izveštaj o plasmanu, pri čemu su učenici prikazani u istom redosledu kao i u polaznoj listi sa poenima.

Da bi se odredio plasman učenika, jasno je da je učenike potrebno sortirati po poenima. Međutim, pošto se ispis traži u originalnom redosledu, ne smemo izgubiti informaciju o njemu.

Drugi mogući način rešavanja je da se napravi sortirana lista poena, a da onda plasman određujemo tako što poene svakog učenika binarnom pretragom tražimo u sklopu sortirane liste poena.

```
  // ucitavamo podatke o učenicima i gradimo pomoćni niz poena
  int n;
  cin >> n;
  vector<pair<string, int>> ucenici(n);
```

```
vector<int> poeni(n);
for (int i = 0; i < n; i++) {
  string ime; int poen;
  cin >> ime >> poen;
  ucenici[i] = make_pair(ime, poen);
  poeni[i] = poen;
}

// sortiramo niz poena opadajući
sort(begin(poeni), end(poeni), greater<int>());

// obilazimo sve učenike po redosledu učitavanja
for (int i = 0; i < ucenici.size(); i++) {
  string ime = ucenici[i].first;
  int poen = ucenici[i].second;

  // binarnom pretragom određujemo plasman
  auto it = lower_bound(begin(poeni), end(poeni), poen,
                        greater<int>());
  int plasman = distance(begin(poeni), it) + 1;

  // ispisujemo ime i plasman tekućeg učenika
  cout << ime << " " << plasman << endl;
}
```

Recimo i da bi se u slučaju da nekoliko takmičara ima isti broj poena ovim načinom dobilo da svi takmičari koji imaju isti broj poena dele isti rang, a da je rang takmičara iza njih određen ispravno (na primer, imali bismo dva druga mesta, nijedno treće i onda četvrto).

Složenost faze sortiranja je $O(n \log n)$ isto kao i naredne faze u kojoj se izvršava n binarnih pretraga čije je vreme izvršavanja $O(\log n)$, pa je i ukupna složenost $O(n \log n)$.

Drugi način rešavanja je da se napravi niz struktura (ili torki) koje će sadržati ime učenika, broj njegovih poena, i originalno mesto u nizu i da se onda taj niz sortira po broju poena, opadajuće, da se zatim na osnovu obilaska tako sortiranog niza popuni niz u kome se čuvaju imena i plasmani učenika i da se onda taj niz redom ispiše.

```
// gradimo niz torki u kojima se čuvaju poeni učenika, njihovi
// redni brojevi i imena i prezimena
int n;
cin >> n;
vector<tuple<int, int, string>> ucenici(n);
for (int i = 0; i < n; i++) {
  string ime; int poeni;
  cin >> ime >> poeni;
  ucenici[i] = make_tuple(-poeni, i, ime);
}
```

```
// sortiranje teče leksikografski, prvo po prvoj komponenti,
// što je negiran broj poena, pa se ovde vrši sortiranje po broju
// poena, nerastuće
sort(begin(ucenici), end(ucenici));

// svakom imenu i prezimenu pridružujemo plasman, a redosled
// u ovom vektoru odgovara originalnom redosledu unosa
vector<tuple<string, int>> plasman(n);
for (int i = 0; i < n; i++)
  plasman[get<1>(ucenici[i])] =
      make_tuple(get<2>(ucenici[i]), i + 1);

// ispisujemo imena i plasmane učenika
for (int i = 0; i < n; i++)
  cout << get<0>(plasman[i]) << " "
       << get<1>(plasman[i]) << endl;
```

Složenost faze sortiranja je $O(n \log n)$, dok je složenost druge faze $O(n)$, uz malo veće zauzeće memorije i komplikovaniju implementaciju nego kada smo upotrebljavali binarnu pretragu. Ukupna složenost je i dalje $O(n \log n)$.

Optimizacija binarnom pretragom

U mnogim situacijama binarna pretraga se može upotrebiti za rešavanje optimizacionih problema.

Štucajući podniz

Problem: Ako je s niska, onda je s^n niska koja se dobija ako se svako slovo ponovi n puta, npr. $(xyz)^3$ je $xxxyyyzzz$. Napisati program koji određuje najveći broj n takav da je s^n podniz niske t (to znači da se sva slova niske s^n javljaju u niski t, u istom redosledu kao u s^n, ali ne obavezno uzastopno).

```
bool jeStucajuciPodniz(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 (jeStucajuciPodniz(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 (celi brojevi). Testera se može podešavati takođe sa preciznošću jednog metra.

Jedno rešenje problema se može zasnovati na binarnoj pretrazi po rešenju tj. po traženju optimalne vrednosti korišćenjem binarne pretrage. 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 $O(n \log M)$. Binarnu pretragu možemo implementirati ručno.

```
int testera(vector<int>& visine, int potrebno) {
  // optimalna visina je negde između 0 i visine najvišeg drveta
  int od_visina = 0;
  int do_visina = *max_element(begin(visine), end(visine));
  // binarnom pretragom pronalazimo najveću visinu za koju
  // koju se dobija dovoljna količina drveta (to je poslednji
  // element niza visina koji zadovoljava uslov)
  while (od_visina <= do_visina) {
    int visina = od_visina + (do_visina - od_visina) / 2;

    // količina drveta koja se naseče kada je testera postavljena
    // na visinu visina
    int naseceno = 0;
    for (int v : visine)
      if (v >= visina)
```

```
        naseceno += v - visina;

    // proveravamo da li je to dovoljna količina i u zavisnosti
    // od toga biramo odgovarajuću polovinu intervala za dalju
    // obradu
    if (naseceno >= potrebno)
      od_visina = visina + 1;
    else
      do_visina = visina - 1;
  }
  return do_visina;
}
```

Druga ideja za rešenje zadatka je da se pronađe najviša visina testere koja je jednaka visini nekog drveta sa kojom se dobija dovoljno drveta. Neka je to visina H i neka se njom dobija količina $D \geq D_p$, gde je D_p potrebna količina i neka se na toj visini seče k stabala. Nama je potrebno da podignemo testeru još malo tako da se višak od $D - D_p$ eliminiše. Sigurni smo da će podizanje da bude manje od visine narednog drveta po visini, jer kada se testera digne na visinu narednog drveta, nećemo imati dovoljno nasečenog drveta (jer je H najviša visina drveta za koju je nasečeno dovoljno). Ovo znači da se tim finim podešavanjem visine neće promeniti broj drva koji se seku. Dakle, ako je $H + h$ manje od visine sledećeg drveta, tada će povećanje visine sa H na $H + h$ metara smanjiti isečenu količinu drveta za $k \cdot h$ metara (jer testera i dalje seče istih k stabala). Dakle, potrebno je pronaći maksimalnu vrednost h takvu da je $D - h \cdot k \geq D_p$, tj. da je $h \leq \frac{D-D_p}{k}$, a to je vrednost $\left\lfloor \frac{D-D_p}{k} \right\rfloor$. Visina je tada $H + h$.

Ostaje još pitanje kako odrediti količinu drveta koja se dobija kada se testera postavi na visinu drveta na poziciji k. Jedan način je da se ta količina svaki put iznova računa, no to bi vodilo kvadratnoj složenosti. Bolje rešenje je da se niz sortira i da se zatim količina računa inkrementalno. Spuštanjem sekire na visinu drveta na poziciji k od svakog od prvih k drveta odečeno je parče od $h_k - h_{k-1}$ metara, pa ako znamo količinu drveta koja se iseče kada je testera na visini drveta na poziciji $k-1$, količinu drveta kada je testera na visini drveta na poziciji k možemo veoma jednostavno dobiti uvećavanjem te količine za $k \cdot (h_k - h_{k-1})$.

```
int testera(vector<int>& visina, int potrebno) {
  // sortiramo drva po visini, opadajuće
  sort(visina.begin(), visina.end(), greater<int>());

  // nasečena količina drva
  int naseceno = 0;
  // broj drva koja sečemo
  int k;
  // spuštamo testeru do visine narednog narednog drveta
  for (k = 1; k < n; k++) {
    // računamo koliko je drveta nasečeno ako je testera na visini
```

```
    // drveta na poziciji k - testera tada zaseca tačno k stabala
    naseceno += (visina[k-1] - visina[k]) * k;
    // prvi put kada nasečemo potrebnu količinu prekidamo petlju
    if (naseceno >= potrebno)
      break;
  }

  // popravljamo malo visinu (fino štelujući visinu između visine
  // drveta na poziciji k-1 i drveta na poziciji k) tako da uzmemo
  // što manje, ali opet dovoljno drveta
  return visina[k] + (naseceno - potrebno) / k;
}
```

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.

Jednostavnosti radi ulazne podatke možemo upamtiti kao globalne promenljive.

```
// broj sastojaka potrebnih za jelo
int brojSastojaka;
// dostupan iznos novca
int dostupanIznosNovca;
// potrebna i već dostupna gramaža svakog sastojaka za jelo
vector<int> potrebnoGrama, dostupnoGrama;
// gramaža i cena malog pakovanja
vector<int> maloGrama, maloCena;
// gramaža i cena velikog pakovanja
vector<int> velikoGrama, velikoCena;
```

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 (dobijamo je tako što od proizvoda broja porcija i grama potrebnih za jednu porciju oduzmemo broj grama koji već imamo na raspolaganju). Pod pretpostavkom da na neki način umemo da izračunamo najmanju cenu za koju je moguće kupiti tu količinu sastojka (tim pitanjem ćemo se pozabaviti uskoro), 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 ako i samo ako je ta ukupna minimalna cena manja ili jednaka iznosu novca koji imamo na raspolaganju.

```
// provera da li se može napraviti dati broj porcija
int mozeSeNapraviti(int brojPorcija) {
  // izračunavamo ukupnu cenu potrebnu da se kupe svi
  // potrebni sastojci
  int ukupnaCena = 0;
  for (int i = 0; i < n; i++) {
    // potreban broj grama tekućeg sastojka za dati broj porcija
    int potrebno = brojPorcija * potrebnoGrama[i] - postojiGrama[i];
    // izračunavamo minimalnu cenu potrebnu da se dokupi dovoljna
    // gramaža
    // tog sastojka i ažuriram ukupnu cenu
    ukupnaCena += minCenaZaKolicinu(malaCena[i], malaKolicina[i],
                          velikaCena[i], velikaKolicina[i],
                          potrebno);
  }
  // dati broj porcija se može napraviti samo ako je ukupna cena
  // svih sastojaka manja od dostupnog iznosa novca
  return ukupnaCena <= dostupanIznosNovca;
}
```

Vratimo se pitanju određivanja minimalne cene potrebne za datu gramažu nekog sastojka. Pretpostavljamo da znamo gramažu i cenu malog i gramažu i cenu velikog pakovanja. Iako se ovaj problem linearnog programiranja može rešiti efikasnije, prikazaćemo jednostavno rešenje grubom silom. 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.

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

// određuje najmanju cenu potrebnu da se nabavi potrebna gramaža
// nekog sastojka ako su poznate gramaža i cena velikog i malog
// pakovanja koje možemo da kupujemo
int minCena(int maloCena, int maloGrama,
            int velikoCena, int velikoGrama,
            int potrebnoGrama) {
  // koliki je broj velikih pakovanja dovoljan da se dobije
  // potreban broj grama
```

```
  int najviseVelikih = ceilFrac(potrebnoGrama, velikoGrama);
  // krećemo od toga da kupujemo samo velika pakovanja
  int min = najviseVelikih * velikaCena;
  // probamo sve mogućnosti i svaki put kupujemo jedno veliko
  // pakovanje manje
  for (int velikih = najviseVelikih - 1; velikih >= 0; velikih--) {
    // izračunavamo koliko je još potrebno dokupiti grama
    int potrebnaKolicinaMalih =
           potrebnaKolicina - velikih * velikaKolicina;
    // izračunavamo broj potrebnih malih pakovanja za tu gramažu
    int malih = ceilFrac(potrebnaKolicinaMalih, malaKolicina);
    // trenutna cena
    int cena = velikih * velikaCena + malih * malaCena;
    // ažuriramo minimum ako je to potrebno
    if (cena < min)
      min = cena;
  }
  // vraćamo pronađenu najmanju cenu
  return min;
}
```

Kada imamo funkciju provere, optimimum tražimo binarnom pretragu. Gornju granicu možemo grubo proceniti i na osnovu ulaznih parametara (na primer, za svaki proizvod lako možemo odrediti najveću količinu porcija za kojih imamo dovoljno tog proizvoda pod pretpostavkom da smo ceo novčani iznos odvojili samo na taj proizvod). 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 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;
```

Čas 8.1, 8.2, 8.3 - tehnika dva pokazivača

Objedinjavanje sortiranih elemenata

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.

```
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++];
}
```

Napomenimo i da 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.

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 $O(n \log n)$, međutim, može se i efikasnije od toga.

Definišimo rekurzivnu funkciju koja izračunava koliko parova čiji je zbir s ima u delu niza čije su pozicije u intervalu $[l, d]$. Traženi broj parova jednak je vrednosti te funkcije za interval $[0, n-1]$, tj. za poziv $l = 0$, $d = n - 1$.

- Ako je $l \geq d$ tada je interval jednočlan ili prazan i u njemu ne može da postoji ni jedan par (a_i, a_j) takav da je $i < j$, pa funkcija treba da vrati nulu.
- Ako je $a_l + a_d > s$, tada je broj parova jednak broju parova u intervalu $[l, d-1]$. Zaista, iz razmatranja smo eliminisali sve parove oblika $(a_{l'}, a_d)$,

za $l \leq l' < d$, međutim, ni jedan od tih parova ne može imati zbir s, jer je niz sortiran i važi da je $a_{l'} + a_d \geq a_l + a_d > s$.

- Ako je $a_l + a_d < s$, tada je broj parova jednak broju parova u intervalu $[l+1, d]$. Zaista, iz razmatranja smo eliminisali sve parove oblika $(a_l, a_{d'})$, za $l < d' \leq d$, međutim, ni jedan od tih parova ne može imati zbir s, jer je niz sortiran i važi da je $a_l + a_{d'} \leq a_l + a_d < s$.
- Na kraju, ako je $a_l + a_d = s$, tada je broj parova za jedan veći od broja parova u intervalu $[l+1, d-1]$. Naime, iz razmatranja eliminišemo sve parove oblika $(a_{l'}, a_d)$ za $l < l' < d$, sve parove oblika $(a_l, a_{d'})$ za $l < d' < d$ i par (a_l, a_d). Zbir ovog poslednjeg jeste s, dok nijedan od ostalih eliminisanih parova ne može da ima zbir s. Naime, pošto je niz sortiran i svi elementi su različiti, on je strogo rastući, pa je $a_{l'} > a_l$ i važi da je $a_{l'} + a_d > a_l + a_d = s$. Slično je i $a_{d'} < a_d$ i važi da je $a_l + a_{d'} < a_l + a_d = s$.

```
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);
}

int brojParova(const vector<int>& a, int s) {
  int n = a.size();
  return brojParova(a, s, 0, n-1);
}
```

Naravno, moguće je jednostavno se osloboditi rekurzije. Članove datog zbira možemo tražiti polazeći sa oba kraja niza. Obilazimo niz sa oba kraja levog $(l = 0)$ i desnog $(d = n - 1)$. Uporedimo $a_l + a_d$ sa s.

- Ako je $a_l + a_d > s$ potrebno je smanjiti zbir para elemenata, to postižemo uzimanjem manjeg elementa u nizu, niz je sortiran u rastućem poretku pa prelazimo na sledeći element u desnom delu niza (d umanjujemo za 1).
- Ako je $a_l + a_d < s$ potrebno je povećati zbir para elemanata, to se postiže uzimanjem većeg elementa u nizu, niz je sortiran u rastućem poretku pa prelazimo na sledeći element u levom delu niza ($ l $ uvećavamo za 1).
- Ako je $a_l + a_d = s$ uvećamo broj traženih parova, i prelazimo na sledeći element u levom delu niza (l uvećavamo za 1) i na sledeći element u desnom delu niza (d umanjujemo za 1).

Proces nastavljamo dok ne obiđemo ceo niz, to jest dok je $l < d$.

```
int brojParovaDatogZbira(const vector<int>& a, int s) {
  int brojParova = 0;
```

```
  int levo = 0, desno = n - 1;
  while (levo < desno)
    if (a[levo] + a[desno] > s)
      desno--;
    else if (a[levo] + a[desno] < s)
      levo++;
    else {
      brojParova++;
      levo++;
      desno--;
    }
  return brojParova;
}
```

Dokažimo korektnost prethodnog postupka. Posmatrajmo skup $S_{l,d}$ koji sadrži sve parove (a_i, a_j) takve da je $0 \leq i < l$ i da je $d < j < n$. Invarijanta prethodne petlje biće to da:

- promenljiva koja čuva tekući broj parova (označimo je sa b) čuva broj parova skupa $S_{l,d}$ takve da je $a_i + a_j = s$,
- za svako $0 \leq i < l$ važi da je $a_i + a_d < s$ i da za svako $d < j < n$ važi da je $a_l + a_j > s$.

Pre ulaska u petlju važi da je $l = 0$ i $d = n - 1$. Tada nema indeksa i takvog da važi $0 \leq i < l$ niti indeksa j takvog da važi $d < j < n$, pa je $S_{l,d}$ prazan. Pošto je $b = 0$, oba dela invarijante važe.

Pretpostavimo da invarijanta važi pri ulasku u telo petlje i dokažimo da je izvršavanje tela petlje održava.

Pretpostavimo prvo da je $a_l + a_d > s$. Tada d umanjujemo za 1 ne menjajući pri tom broj parova b, tj. nakon izvršavanja tela petlje važi da je $l' = l$, $d' = d - 1$ i $b' = b$. Skup $S_{l',d'} = S_{l,d-1}$ se može razložiti na: (1) skup $S_{l,d}$ i (2) skup skup svih parova (a_i, a_j) takvih da je $0 \leq i < l$ i $j = d$. Broj parova traženog zbira u skupu $S_{l,d}$ jednak je b (na osnovu prvog invarijante), dok se u drugom skupu ne nalazi ni jedan takav par. Zaista, na osnovu drugog dela invarijante znamo da svako $0 \leq i < l$ važi da je $a_i + a_d < s$, pa među parovima drugog skupa ne može biti ni jedan koji ima zbir jednak s. Pošto je $b' = b$, prvi deo invarijante ostaje očuvan. Potrebno je da pokažemo i da drugi deo invarijante ostaje očuvan. Prvo, treba da dokažemo da za svako $0 \leq i < l'$ važi da je $a_i + a_{d'} < s$. Pošto je $l' = l$, na osnovu drugog dela invarijante znamo da za sve takve indekse i važi da je $a_i + a_d < s$, a pošto je niz sortiran i pošto su mu elementi različiti, važi da je $a_{d'} = a_{d-1} < a_d$, pa je $a_i + a_{d'} = a_i + a_{d-1} < a_i + a_d < s$. Treba da dokažemo i da za svako $d' < j < n$, važi da je $a_{l'} + a_j > s$. Na osnovu drugog dela invarijante to važi za sve $d < j < n$. Pošto je $l' = l$ i $d' = d - 1$, ostaje samo još da se dokaže da taj uslov važi za d, tj. samo da se dokaže da važi da je $a_l + a_d > s$, no to važi na osnovu pretpostavke (tj. grane koju trenutno analiziramo).

Veoma slično se pokazuje da se u slučaju $a_l + a_d < s$ povećanjem broja l i ne menjanjem broja b invarijanta održava.

Na kraju, ostaje slučaj kada je $a_l + a_d = s$. U tom slučaju se vrši uvećanje broja l, umanjenje broja d i uvećanje broja b, tj. važi da je $l' = l + 1$, $d' = d - 1$ i $b' = b + 1$. Skup $S_{l',d'} = S_{l+1,d-1}$ se može razložiti na: (1) skup $S_{l,d}$, (2) skup svih parova (a_i, a_j) takvih da je $0 \leq i < l$, $j = d$, (3) skup svih parova takvih da je $i = l$, $d < j < n$ i (4) skup koji sadrži samo par (a_l, a_d). Na osnovu prvog dela invarijante važi da u skupu $S_{l,d}$ ima b parova čiji je zbir s. Pošto je $a_l + a_d = s$, (a_l, a_d) je još jedan traženi par. Ostaje još da pokažemo da u preostala dva skupa ne postoji ni jedan par čiji je zbir s. Zaista, skup svih parova takvih da je $0 \leq i < l$, $j = d$ i $a_i + a_j = s$, je prazan, jer na osnovu drugog dela invarijanate znamo da je u svim tim parovima $a_i + a_j < s$. Analogno, na osnovu drugog dela invarijante dokazujemo da je prazan i skup svih parova takvih da je $i = l$, da je $d < j < n$ i $a_i + a_j = s$, jer za sve te parove važi da je $a_i + a_j > s$. Dakle, prvi deo invarijante ostaje očuvan. Potrebno je još dokazati da je očuvan i drugi deo invarijante. Potrebno je dokazati da je za za svako $0 \leq i < l'$ važi da je $a_i + a'_d < s$. Na osnovu drugog dela invarijante znamo da za svako $0 \leq i < l$ važi da je $a_i + a_d < s$. Pošto je niz sortiran i svi su mu elementi različiti i pošto je $d' = d - 1$, važi da je $a_{d'} < a_d$. Zato je $a_i + a_{d'} < a_i + a_d < s$. Pošto je $l' = l + 1$, ostaje da se to dokaže još da je $a_l + a_{d'} < s$. No znamo da je $a_l + a'_d < a_l + a_d = s$. Analogno se dokazuje još da za svako $d' < j < n$ važi $a'_l + a_j > s$.

Kraj petlje nastupa kada je $l \geq d$. Skup svih parova (a_i, a_j) takvih da je $0 \leq i < j < n$ se može razložiti na: (1) skup svih parova (a_i, a_j) takvih da je $j \leq d$, (2) skup $S_{l,d}$ tj. skup svih parova (a_i, a_j) takvih da je $0 \leq i < l$ i $d < j < n$ i (3) skup svih parova takvih da je $l \leq i$. U prvom i trećem skupu nema ni jedan par čiji je zbir jednak s. Zaista, ako je $j \leq d$, tada važi da je $0 \leq i < j \leq d \leq l$. Na osnovu drugog dela invarijante znamo da tada važi da je $a_i + a_d < s$, a pošto je niz sortiran važi i da je $a_j \leq a_d$, pa je $a_i + a_j \leq a_i + a_d < s$. Ako važi da je $l \leq i$, tada važi i da je $d \leq l \leq i < j < n$. Na osnovu drugog dela invarijante znamo da je $a_l + a_j > s$, pa pošto je niz sortiran važi da je $a_l \leq a_i$ i zato je $a_i + a_j \geq a_l + a_j > s$. Na osnovu prvog dela invarijante znamo da je broj b jednak broju parova iz drugog skupa, što je na osnovu prethodnog ukupan broj traženih parova.

Pošto se u svakom koraku razlika između d i l smanji bar za 1 (nekada i za 2), ukupan broj koraka ne može biti veći od n, pa je složenost $O(n)$.

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 (a_i, a_j, a_k) za $i < j < k$, tj. da se upotrebe tri ugnežđene petlje u čijem se telu proverava da li je $a_i + a_j + a_k =$

0 i ako jeste, da se uveća brojač. Složenost ovog algoritma odgovara broju trojki i jednaka je $O(n^3)$, što je nedopustivo veliko.

Pretpostavimo da se elementi u trojkama javljaju u istom redosledu u kom se javljaju u nizu. Da bi se neka trojka (a_i, a_j, a_k), $i < j < k$ 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) {
  // ukupan broj trojki čiji je zbir 0
  int brojTrojki = 0;
  for (int i = 0; i < n - 2; i++) {
    // izracunavamo broj parova u delu niza [i+1, n) ciji je zbir -a[i]
    // posto je ceo niz sortiran, sortiran je i taj deo
    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;
}
```

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. Složenost ovakvog algoritma je $O(n^2)$.

Kao i u mnogim problemima pretrage, sortiranje niza može dovesti do efikasnijih rešenja.

Jedno rešenje može biti zasnovano na binarnoj pretrazi. Da bi važilo $a_j - a_i = r$ potrebno i dovoljno je da važi da je $a_j = a_i + r$. Pošto je $r > 0$, a niz je sortiran neopadajuće, a_j se, ako postoji, nalazi na pozicijama iza i. Dakle, za svaki element niza a_i u delu niza iza njega proveravamo da li postoji element $a_i + r$. Pošto je niz sortiran, to možemo efikasno izvršiti binarnom pretragom. Da u nizu nema ponavljanja, za svaki pronađeni element $a_i + r$ uvećavali bismo brojač parova za 1. Međutim, pošto može biti ponavljanja potrebno je da izračunamo broj pojavljivanja elementa $a_i + r$ i da brojač uvećamo za tu vrednost (jer se svako pronađeno pojavljivanje kada se ukombinuje sa elementom a_i daje jedan takav par). Ako se element a_i pojavljuje više puta, tada možemo uštedeti broj poziva binarne pretrage tako što izračunamo broj pojavljivanja elementa a_i (neka je to b_i), elementa $a_j = a_i + r$ (neka je to b_j) i uvećati brojač za $b_i \cdot b_j$.

U jeziku C++ broj pojavljivanja elementa u sortiranom nizu možemo najbolje odrediti bibliotečkom funkcijom `equal_range`.

```
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  // sortiramo niz
  sort(begin(a), end(a));

  // broj parova date razlike
  int brojParova = 0;
  // broj pojavljivanja elementa na tekućoj poziciji i
  int brojPojavljivanja_ai = 1;
  for (int i = 1; i < n; i++) {
    // ako je element na poziciji i jednak prethodnom,
    // uvećavamo broj pojavljivanja elementa na poziciji i
    if (a[i] == a[i-1])
      brojPojavljivanja_ai++;
    else {
      // binarnom pretragom određujemo sve elemente u delu niza od
      // pozicije i jednake elementu a[i-1] + razlika
      // oni sa elementom a[i-1] daju razliku razlika
      auto range =
        equal_range(next(begin(a), i+1), end(a), a[i-1] + razlika);
      // broj pojavljivanja tih elemenata
      int brojPojavljivanja_aj = distance(range.first, range.second);
      // uvećavamo broj parova
      brojParova += brojPojavljivanja_ai * brojPojavljivanja_aj;
      // u koraku petlje prelazimo na naredni element, pa
      // inicijalizujemo njegov broj pojavljivanja
      brojPojavljivanja_ai = 1;
    }
  }
  return brojParova;
}
```

Složenost obe faze (i sortiranja i binarne pretrage) je $O(n \log n)$. Možemo i efikasnije od ovoga, koristeći tehniku dva pokazivača. Pretpostavimo prvo da u

nizu nema duplikata i definišimo rekurzivnu funkciju koja određuje broj parova date razlike.

Funkcija određuje koliko parova date razlike r postoji u intervalu $[i, n)$, a invarijanta poziva ove funkcije je da je $a_{j-1} - a_i < r$. U prvom pozivu je $j = 1$ i $i = 0$, pa je $a_{j-1} - a_i = a_0 - a_0 = 0 < r$.

- Ako je $j = n$, tada u intervalu ne postoji ni jedan par date razlike. Zaista, na osnovu invarijante važi da je $a_{n-1} - a_i < r$. Pošto je niz sortiran, i povećanjem i i smanjivanjem j razlika se smanjuje. Zato parovi brojeva unutar intervala $[i, n)$ imaju manju razliku od r.
- Ako je $a_j - a_i < r$, tada znamo da u intervalu $[i, j]$ ne postoji ni jedan par brojeva čija je razlika jednaka r (jer je razlika elemenata unutar intervala uvek manja neko razlika krajnjih elemenata). Invarijanta je zadovoljena za par $(i, j + 1)$ pa naredni poziv vršimo za taj interval.
- Ako je $a_j - a_i > r$, tada ni jedan par $a_{j'} - a_i$ za $i < j' < n$ nema razliku r. Na osnovu invarijante znamo da je $a_{j-1} - a_i$ manje od r, pa pošto je niz sortiran to važi i za sve elemente $i < j' < j$. Pošto je $a_j - a_i > r$ i pošto je niz sortiran povećanjem j se povećava razlika, pa su razlike za $j \leq j' < n$ veće od r. Zato se u intervalu $[i, n)$ svi eventualni parovi čija je razlika r nalaze u intervalu $[i + 1, n)$. Još moramo dokazati da tada invarijanta važi tj. da je $a_{j-1} - a_{i+1} < r$, međutim to važi jer je niz sortiran i važi $a_i < a_{i+1}$, a na osnovu invarijante je važilo da je $a_{j-1} - a_i < r$.
- Na kraju, ako je $a_j - a_i = r$, tada smo pronašli jedan par. Pošto smo pretpostavili da u nizu nema duplikata i da je niz sortiran, a_i ne može biti član ni jednog drugog para sa razlikom r u intervalu $[i, n)$ - pošto je niz sortiran pomeranjem umanjenika nalevo razlika se smanjuje, a pomeranjem nadesno, ona se povećava. Dakle, svi eventualni parovi čija je razlika r nalaze se u intervalu $[i + 1, n)$. Naredni poziv se može izvršiti za argumente $(i + 1, j + 1)$. Zaista, invarijanta je zadovoljena jer je $a_{j+1-1} - ai + 1 = a_j - a_{i+1} < a_j - a_i = r$.

Odatle sledi korektnost naredne rekurzivne imiplementacije, koja se veoma jednostavno prevodi u iteraciju.

```
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) {
```

```
  // sortiramo niz
  sort(begin(a), end(a));
  return brojParovaDateRazlike(a, razlika, 0, 1);
}
```

Ako se elementi u nizu ponavljaju, onda u trenutku kada nađemo prvi par (i, j) takav da je $a_i - a_j = r$, određujemo broj pojavljivanja n_i elementa a_i i broj pojavljivanja n_j elementa a_j, broj parova uvećavamo za $n_i \cdot n_j$ (jer svako pojavljivanje vrednosti a_i možemo iskombinovati sa svakim pojavljivanjem vrednosti a_j) i nakon toga vršimo rekurzivni poziv za par $(i + n_i, j + n_j)$.

```
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  int broj = 0;
  int i = 0, j = 1;
  while (j < n) {
    if (a[j] - a[i] < razlika)
      j++;
    else if (a[j] - a[i] > razlika)
      i++;
    else {
      int ii;
      for (ii = i+1; ii < n && a[ii] == a[i]; ii++)
         ;
      int broj_ai = ii - i;
      i = ii;
      int jj;
      for (jj = j+1; jj < n && a[jj] == a[j]; jj++)
        ;
      int broj_aj = jj - j;
      j = jj;
      broj += broj_ai * broj_aj;
    }
  }
}
```

Još jedan zgodan način implementacije je da kreiramo mapu u kojoj elemente preslikavamo u njihove brojeve pojavljivanja, a onda da originalni algoritam sa dva pokazivača (u ovom slučaju dva iteratora) primenimo na mapu (ključevi u mapi će biti sortirani).

```
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  // preslikavamo svaki element u njegov broj pojavljivanja
  map<int, int> m;
  for (auto x : a)
    m[x]++;

  // određujemo broj elemenata čija je razlika jednaka datoj
  int broj = 0;
  auto i = m.begin(), j = m.begin();
```

```
  while (j != m.end()) {
    if (j->first - i->first < razlika)
      j++;
    else if (j->first - i->first > razlika)
      i++;
    else {
      broj += j->second * i->second;
      j++;
    }
  }
  return broj;
}
```

Složenost prve faze (sortiranja niza ili kreiranja mape) je $O(n \log n)$, dok je druga faza linearne složenosti (kao i uvek kada se koristi tehnika dva pokazivača), pa je ukupna složenost $O(n \log n)$.

Pretraga dvostruko sortirane matrice

Problem: Svaka vrsta i svaka kolona matrice dimenzije $m \times n$ 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.

```
bool sadrzi(int a[MAX][MAX], int m, int n, int x) {
   for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
         if (a[i][j] == x)
            return true;
   return false;
}
```

Složenost najgoreg slučaja nastupa kada elementa nema u matrici i prilično je jasno da iznosi $O(m \cdot n)$.

U prethodnom rešenju uopšte nismo upotrebili činjenicu da su vrste i kolone matrice sortirane. Naredno rešenje koje pada na pamet je da se umesto linearne pretrage svake vrste vrši njihova binarna pretraga. Binarnom pretragom možemo postići da se svaka vrsta dužine n pretraži u $O(\log n)$ koraka, pa je ukupna složenost $O(m \log n)$. Ako je $m > n$, tada je bolje binarnu pretragu primenjivati na kolone, pa bi tada složenost bila $O(n \log m)$.

```
bool sadrzi(int a[MAX][MAX], int m, int n, int x) {
  for (int i = 0; i < m; i++) {
     int l = 0, d = n;
```

```
    while (l <= d) {
       int s = l + (d - l) / 2;
       if (x < a[i][s])
          d = s-1;
       else if (x > a[i][s])
          l = s+1;
       else
          return true;
    }
  }
  return false;
}
```

Naravno, u jeziku C++ na raspolaganju imamo i bibliotečku implementaciju binarne pretrage.

```
bool sadrzi(int a[MAX][MAX], int m, int n, int x) {
  for (int i = 0; i < m; i++)
    if (binary_search(A[i], A[i]+n, x))
      return true;
  return false;
}
```

U prethodnom rešenju smo upotrebili samo činjenicu da su vrste matrice sortirane, međutim, mi znamo i više od toga - sortirane su i vrste i kolone. Jedan način da se to iskoristi je da se binarna pretraga modifikuje tako što se rezultat binarne pretrage prethodne vrste upotrebi da bi se skratila binarna pretraga naredne vrste. Modifikujmo binarnu pretragu tako da nam prilikom pretrage intervala $[0, d]$ vraća prvu poziciju p u vrsti na kojoj se nalazi element koji je veći ili jednak od traženog. Ako takav element ne postoji, binarna pretraga treba da vrati vrednost $d + 1$ i u tom slučaju i u narednoj vrsti pretražujemo interval $[0, d]$. U suprotnom je pozicija manja ili jednaka d i ako se na njoj nalazi traženi element, pretraga se završava. U suprotnom je element na poziciji p veći od traženog. Pošto su kolone sortirane neopadajuće, znamo da je i u narednoj vrsti na poziciji p element koji je veći od x, a pošto su vrste sortirane, i svi elementi iza pozicije p su veći od x. Dakle, narednu vrstu možemo da pretražujemo samo do pozicije $p - 1$.

```
bool sadrzi(int a[MAX][MAX], int m, int n, int x) {
  int d = n - 1;
  for (int i = 0; i < m && d > 0; i++) {
     int l = 0;
     while (l <= d) {
        int s = l + (d - l) / 2;
        if (x <= a[i][s])
           d = s-1;
        else
           l = s+1;
     }
```

```
    if (l < n && a[i][l] == x)
        return true;
    d = l-1;
  }
  return false;
}
```

Međutim, ova optimizacija ne menja složenost najgoreg slučaja. Naime, ako je element koji se traži veći od svih elemenata matrice, sve binarne pretrage će se zaustavljati u stanju $d = n$, tako da će složenost i dalje biti $O(m \log n)$ ili $O(n \log m)$, u zavisnosti od toga da li pretražujemo vrste ili kolone.

Recimo da se i ovo rešenje može pojednostaviti primenom bibliotečkih funkcija jezika C++ (funkcija `lower_bound` vraća prvi element u datom rasponu koji nije manji od datog).

```
bool sadrzi(int a[MAX][MAX], int m, int n, int x) {
  for (int i = 0; i < m && n > 0; i++) {
    auto end = a[i] + n;
    auto it = lower_bound(a[i], end, x);
    if (it != end && *it == x)
      return true;
    n = distance(a[i], it);
  }
  return false;
}
```

Kako ovaj pristup nije dao rešenje željene složenosti, vratimo se korak nazad. Moć binarne pretrage proizilazi iz sortiranosti i mogućnosti da se zahvaljujući tome u svakom koraku značajan broj elemenata eliminiše iz pretrage, a da se ne troši vreme na njihovu analizu. Razmotrimo matricu

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

i zapitajmo se postoje li još neke formacije u matrici koje su sortirane? 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 (npr. `1, 2, 3, 4, 6, 8, 10`, ili `3,4,6,8, ili5,6,7,9). Razmotrimo prvi navedeni zglob. Ukoliko bismo ga pretraživali binarnom pretragom, krenuli bismo od njegove sredine, odnosno od broja4`. Eliminacijom jedne strane zgloba, eliminise se jedna kolona ili jedna vrsta matrice. Potom se ovaj postupak rekurzivno ponavlja.

```
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;
}

bool sadrzi(int a[MAX][MAX], int m, int n, int x) {
   return sadrzi(a, m, n, m-1, 0, x);
}
```

Pošto je rekurzija repna, ona se veoma jednostavno eliminiše.

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

Jasno je da je broj poređenja najviše $m+n$, pa da je složenost $O(m+n)$. Dati primer sugeriše da će pomenuta pretraga uvek ličiti na binarnu, odnosno da će uvek biti eliminisana polovina zgloba. Imajmo u vidu da ako to ne mora biti tako. Na primer, moguće je da se u prvim koracima eliminišu samo kolone, čime se u svakom koraku eliminiše više od polovine zgloba, a da se na kraju vrši linearna pretraga po poslednjoj koloni. Ipak neefikasnost poslednje faze je kompenzovana efikasnošću prethodnih faza pretrage.

Primetimo da se ova implementacija se može podvesti pod tehniku dva pokazivača (jer se i `v` i `k` menaju samo u jednom smeru). Analiza složenosti je mogla biti izvedena i iz te perspektive. Dok je uslov petlje ispunjen vrednost $v+n-k$ je sve vreme nenegativna, kreće od $m+n-1$, i u svakom koraku se smanjuje za 1. Posle najviše $m+n$ koraka ona sigurno postaje negativna, što garantuje da će se petlja završiti. Dakle, složenost algoritma je $O(m+n)$.

Što se tiče vremena izvršavanja, izvršili smo eksperimente na matrici dimenzije 1000×1000, takvoj da je $a_{ij}=5(i+j)$ i u njoj smo tražili sve elemente između 0 i 10000 (u matrici ih ima tačno 1999). Linearna pretraga rezultat vraća nakon oko $3,861$ sekundi. Obična, ručno implementirana, binarna pretraga vraća rezultat nakon $0,343$ sekunde, a ona zasnovana na bibliotečkoj funkciji za oko $0,307$ sekundi. Optimizovanoj binarnoj pretrazi potrebno je oko $0,155$ sekundi (i kada je implementirana ručno i kada je implementirana pomoću bibliotečke

funkcije). Na kraju, rešenje linearne složenosti radi za oko $0,054$ sekunde (i kada je implementirana iterativno i kada je implementirana rekurzivno).

Segmenti niza prirodnih brojeva koji imaju dati zbir

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 tj. važi da iz $i < i' \leq j$ sledi $z_{ij} > z_{i'j}$ i da iz $j < j' < n$ sledi $z_{ij} < z_{ij'}$.

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 (jer je i za svako i' takvo da je $i < i' \leq j = n - 1$ važi $z_{i'j} < z_{ij} < z$, a zato i za svako j' takvo da je $i' \leq j' < j = n-1$ važi da je $z_{i'j'} < z_{i'j} < z$, tako da za svaki interval $[i', j']$ takav da je $i \leq i' \leq j' < n$ važi da je $z_{i'j'} < z$).
- 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 (ako je $z_{ij} > z$, onda takvih segmenata nema). Naime, 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 $z_{(i+1)j'} < z$. Naime, 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.

Jedan način da se na osnovu prethodne analize napravi implementacija je da se u svakom koraku petlje održavaju dve promenljive i i j i promenljiva *zbir*. Obezbedićemo da pri svakom ulasku u telo petlje važi da promenljiva *zbir* čuva tekuću vrednost z_{ij} i da je za svako $j' < j$ ispunjeno da je $z_{ij'} < z$. Ako se i i j inicijalizuju na nulu, tada se *zbir* treba inicijalizovati na a_0, čime se zadovoljava prethodni uslov.

U telu petlje provervamo da li je *zbir* manji od traženog i ako xjeste, uvećavamo vrednost j. Ako j dostigne vrednost n, tada možemo prekinuti petlju i završiti pretragu. Ako je uvećano j manje od n, onda zbir uvećavamo za a_j i prelazimo na naredni korak petlje (uslov koji smo nametnuli da važi pri ulasku u petlju će biti ovim biti zadovoljen).

Ako vrednost promenljive *zbir* nije manja od tražene vrednosti z proveravamo da li joj je jednaka. Ako jeste, prijavljujemo pronađeni interval $[i, j]$. Zatim prelazimo na obradu narednog intervala tako što zbir umanjujemo za vrednost a_i i i uvećavamo za 1 (uslov koji smo nametnuli da važi pri ulasku u petlju će ovim biti opet zadovoljen).

```
void ispisiSegmenteDatogZbira(const vector<int>& a, int trazeniZbir) {
  // granice segmenta
  int i = 0, j = 0;
  // zbir segmenta
  int zbir = a[0];
  while (true) {
    // na ovom mestu vazi da je zbir = sum(ai, ..., aj) i da
    // za svako i <= j' < j vazi da je sum(ai, ..., aj') < trazeniZbir

    if (zbir < trazeniZbir) {
      // prelazimo na interval [i, j+1]
      j++;
      // ako takav interval ne postoji, zavrsili smo pretragu
      if (j >= n)
         break;
      // izracunavamo zbir intervala [i, j+1] na osnovu zbira intervala [i, j]
      zbir += a[j];
    } else {
      // ako je zbir jednak trazenom, vazi da je sum(ai, ..., aj) = trazeniZbir
      // pa prijavljujemo interval
      if (zbir == trazeniZbir)
          cout << i << " " << j << endl;
      // prelazimo na interval [i+1, j]
      // izracunavamo zbir intervala [i+1, j] na osnovu zbira intervala [i, j]
        zbir -= a[i];
        i++;
    }
  }
}
```

Još jedno veoma elegantno rešenje zadatka se svodi na prefiksne zbirove i činjenicu da je zbir svakog segmenta razlika dva zbira prefiksa. Ako znamo zbirove svih prefiksa, pošto su brojevi pozitivni, prirodni niz tih zbira prefiksa će biti sortiran rastuće. Stoga je potrebno da u rastućem nizu pronađemo sve elemente kojima je razlika jednaka datom broju, a taj problem smo već uspešno rešili tehnikom dva pokazivača.

```
void ispisiSegmenteDatogZbira(const vector<int>& a, int trazeniZbir) {
```

```
  // izracunavamo parcijalne zbirove elemenata niza
  int n;
  cin >> n;
  vector<int> S(n+1);
  S[0] = 0;
  for (int i = 0; i < n; i++)
    S[i+1] = S[i] + a[i];

  // u sortiranom nizu parcijalnih zbirova trazimo da li postoje dva
  // elementa cija je razlika jednaka trazenom zbiru
  int l = 0, d = 1;
  while (d <= n) {
    if (S[d] - S[l] < trazeniZbir) {
      d++;
    } else if (S[d] - S[l] > trazeniZbir) {
      l++;
    } else {
      cout << l << " " << (d - 1) << endl;
      l++;
    }
  }
}
```

Najkraća podniska koji 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 (one su određeni indeksima $0 \leq i \leq j \leq n$), 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 pretregu grubom silom je veoma jednostavno implementirati, ali njena složenost je $O(n^3 \cdot m)$ gde je n dužina teksta, a m broj karaktera u skupu S. Parametar m ima malu vrednost, ali dimenzija n može značajno da raste, pa je poželjno pronaći bolji algoritam.

Mogući korak optimizacije može biti da se primeti 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. Na primer, ako se u niski `xCxxBxxBxxAxxBxxCxxxxBxAxAxCxxxBx` traži podniska u kome se javljaju slova iz skupa `ABC`. Nakon pronalaska podniske `CxxBxxBxxA`, koja sadrži sve potrebne karaktere, nema potrebe razmatrati njene sufikse (na primer, segment `CxxBxxBxxAxxB`). Dakle prva pronađena ispravna podniska koja počinje na poziciji i ujedno je i najkraća ispravna podniska pronađena na toj poziciji. Zato je nakon odmah nakon pronalaska prve ispravne podniske i ažuriranja vrednosti najmanje dužine moguće prekinuti unutrašnju petlju (naredbom

`break`). Ova optimizacija ne popravlja asimptotsku složenost najgoreg slučaja.

```
// proverava da li niska T[i, j] sadrzi karakter c
bool podniskaSadrziKarakter(const string& T, int i, int j, char c) {
  for (int k = i; k <= j; k++)
    if (T[k] == c)
      return true;
  return false;
}

// niska koja se pretražuje i skup traženih karaktera S
int najkracaPodniska(const string& niska, const string& S) {
  // dužina najkraće do sada pronađene niske koja sadrži sve
  // karaktere iz S
  int min_duzina = numeric_limits<int>::max();

  // obrađujemo sve pozicije početka podniske
  for (int i = 0; i < niska.size(); i++) {
    // preskačemo karaktere koji nisu u skupu S
    // jer najkraća podniska mora da počne sa karakterom iz S
    if (S.find(T[i]) == string::npos) continue;

    // obrađujemo sve pozicije kraja podniske koja počinje na poziciji i
    for (int j = i; j < T.size(); j++) {
      // proveravamo da li podniska niska[i, j] sadrži sve karaktere iz S
      bool nedostaje = false;
      for (char c : S)
        if (!podniskaSadrziKarakter(T, i, j, c)) {
           nedostaje = true;
           break;
        }
      if (!nedostaje) {
        int duzina = j - i + 1;
        if (duzina < min_duzina)
          min_duzina = duzina;
        break;
      }
    }
  }

  if (min_duzina != numeric_limits<int>::max())
    return min_duzina;
  else
    return -1;
}
```

Primetimo dalje 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 (reći ćemo da su samo ti karakteri relevantni), pa ćemo u prvoj fazi samo izgraditi niz (vektor, listu) 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 $O(n \cdot m)$ (pošto je m mali broj, ovo je praktično linearno, a može se sniziti i na $O(m + n)$ ako bi se skup S predstavio svojom karakterističnom funkcijom - asocijativnim nizom 26 logičkih vrednosti).

Proveru da li se svi karakteri iz skupa S javljaju u segmentu imeđu 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 (a pošto niska S po uslovima zadatka nema ponovljenih karaktera, broj elemenata skupa S jednak je dužini niske). 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 (ako skup karaktera predstavimo nizom ili bibliotečkim skupom, jer je ukupan broj mogućih karaktera samo 26). 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 koju na njoj počinje, pronalazimo najkraću podnisku koja se na njoj završava (dualno, mogli bismo da tražimo najkraće podniske koji počinju na svakoj poziciji, kao što smo to radili u prethodnom algoritmu, ali da bismo dobili efikasniji algoritam, pozicije bi trebalo obrađivati sa desna na levo). 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. Opet imamo jednostavni argument da će najkraća podniska 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. Ključne opaske za efikasno određivanje najkraće ispravne podniske koji se završava na nekoj relevantnoj poziciji su to da ćemo uvek 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 (zaista, prefiks te podniske sve do sledećeg karaktera iz skupa S se može ukloniti i opet će se dobiti ispravna podniska tj. podniska koji sadrži sve potrebne karaktere).

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. Ta podniska mora biti najkraća od svih ispravnih podniski koje se završavaju na trenutnoj poziciji jer bi se njegovim izbacivanjem izgubilo svojstvo da podniska sadrži sve karaktere iz skupa S tj. podniska ne bi više bila ispravna.

U primeru `xCxxBxxBxxAxxBxxCxxxxBxAxAxCxxxBx` razmatrali bismo podnisku `C`, zatim `CxxB`, zatim `CxxBxxB` i zatim `CxxBxxBxxA` koja sadrži sve potrebne karaktere (ona je najkraća od svih podniski koji se završavaju na toj poziciji prvog karaktera `A`). Prelaskom na sledeću relevantnu poziciju dobila bi se podniska `CxxBxxBxxAxxB` koja je najkraća od svih koje se na njoj završavaju (zato što se početno `C` javlja samo jedan put i ne sme se ukloniti). Sledeća pozicija je pozicija narednog slova `C`. Konstrukciju najkraće podniske koji se završava na toj poziciji započinjemo tako što proširimo prethodnu podnisku nadesno i dobijamo `CxxBxxBxxAxxBxxC`. Zatim uklanjamo početno `C` i karaktere `x` iza njega (jer se `C` javlja u podniski i kasnije), čime dobijamo `BxxBxxAxxBxxC`, zatim uklanjamo i `Bxx` (jer se `B` još dva puta javlja u podniski) i ponovo uklanjamo `Bxx` (jer se `B` još jednom javlja u podniski). Kada se dođe do podniske `AxxBxxC` ona je najkraća koja se završava na poziciji tog slova `C` jer se `A` javlja samo na njegovom početku i ne sme se ukloniti. Istim postupkom dobija se da je najkraća podniska koji se završava na narednoj poziciji `AxxBxxCxxxxB`, na narednoj poziciji je to podniska `CxxxxBxA`, na narednoj poziciji je to `CxxxxBxAxA`, zatim `BxAxAxC`, i na kraju `AxCxxxB`. Pošto smo za svaku relevantnu poziciju pronašli najkraće ispravne podniske, možemo odrediti i dužinu globalno najkraće i to je 7 (tu dužinu imaju podniske `AxxBxxC`, `BxAxAxC` i `AxCxxxB`).

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 umesto skupa karaktera trenutne podniske koji smo koristili u prošlom algoritmu koristićemo mapu koja svaki karakter preslikava u njegov broj pojavljivanja.

Ocenimo sada i složenost ovog algoritma. Spoljna petlja po j prolazi kroz sve relevantne pozicije kojih ima najviše $O(n)$. Ključna opaska za ocenu složenosti je da se unutrašnja petlja (u kojoj se vrši skraćivanje segmenta koji se završava na poziciji j) može izvršiti ukupno $O(n)$ puta (i je sve vreme manje ili jednako j i stalno se uvećava, a nikada ne umanjuje). U okviru petlji vrši se ažuriranje vrednosti u mapi, ali s obzirom na to da ta operacija ima logaritamsku složenost (ili čak konstantnu, pošto je raspon tipa `char` ograničen), a broj karaktera u skupu S je mali, možemo reći da je složenost algoritma $O(n)$, tj. da je algoritam linearan u odnosu na dužinu teksta.

```
// niska koja se pretražuje i skup u kome se nalaze traženi karakteri
int najkracaPodniska(const string& niska, const string& S) {
  vector<int> poz_karaktera_iz_S;
```

```
  for (int i = 0; i < niska.size(); i++)
    // ako se niska[i] nalazi u skupu S
    if (S.find(niska[i]) != string::npos)
      // zapamti njegovu poziciju i
      poz_karaktera_iz_S.push_back(i);

  // najmanja dužina podniske
  int min_duzina = numeric_limits<int>::max();
  // broj pojavljivanja svakog relevantnog karaktera u trenutnoj podniski
  map<char, int> broj_pojavljivanja_u_podniski;
  // trenutna podniska je odredjena pozicijama [i, j]
  vector<int>::const_iterator i, j;
  for (i = j = poz_karaktera_iz_S.begin(); j != poz_karaktera_iz_S.end(); j++){
    // trenutnu podnisku proširujemo do sledeće pozicije j i
    // uvecavamo broj pojavljivanja karaktera koji se nalazi na
    // poziciji određenoj sa j (jer se i on sada javlja u podniski)
    broj_pojavljivanja_u_podniski[niska[*j]]++;
    // ako mapa ima isto elemenata kao i skup S, onda su svi elementi
    // skupa S prisutni u podniski
    if (broj_pojavljivanja_u_podniski.size() == S.size()) {
      // tražimo najkraću podnisku koji se završava na poziciji
      // određenoj sa j tako što podnisku skraćujemo sa leve strane
      // dok god se prvi karakter podniske javlja više puta u njemu
      while (broj_pojavljivanja_u_podniski[niska[*i]] > 1) {
        // podnisku skraćujemo i uklanjamo sve karaktere sa početka sve do
        // sledeceg karaktera iz skupa S
        broj_pojavljivanja_u_podniski[niska[*i]]--;
        i++;
      }
      // izračunavamo dužinu trenutne podniske
      int duzina = *j - *i + 1;
      // ažuriramo minimum ako je to potrebno
      if (duzina < min_duzina)
        min_duzina = duzina;
    }
  }
  // prijavljujemo rezultat
  if (min_duzina != numeric_limits<int>::max())
    return min_duzina;
  else
    return -1;
}
```

Čas 9.1, 9.2, 9.3 - dekompozicija (divide-and-conquer)

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 $O(n \log n)$, jednačina $T(n) = 2T(n/2) + O(1)$, $T(0) = O(1)$ čije je rešenje $O(n)$ ili jednačina $T(n) = 2T(n/2) + O(\log n)$, $T(0) = O(1)$ 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 $O(n^2)$. Osnovni primere te tehnike su sortiranje objedinjavanjem (engl. Merge Sort) i brzo sortiranje (engl. Quick Sort).

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 $O(\log n)$ ili $T(n) = T(n/2) + O(n)$, $T(0) = O(1)$, čije je rešenje $O(n)$.

U nastavku ćemo prikazati nekoliko primera zadataka rešenih ovom tehnikom.

Merge-sort

Problem: Implementirati sortiranje niza objedinjavanjem sortiranih polovina.

```
// ucesljava deo niza a iz intervala pozicija [i, m] i deo niza b iz
// intervala pozicija [j, n] koji su vec sortirani tako da se dobije
// sortiran rezultat koji se smesta u niz c, krenuvsi od pozicije k
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++];
}

// sortira deo niza a iz intervala pozicija [l, d] koristeci
// niz tmp kao pomocni
void merge_sort(vector<int>& a, int l, int d, vector<int>& tmp) {
  // ako je segment [l, d] jednoclan ili prazan, niz je vec sortiran
  if (l < d) {
    // sredina segmenta [l, d]
```

```
    int s = l + (d - l) / 2;
    // sortiramo segment [l, s]
    merge_sort(a, l, s, tmp);
    // sortiramo segment [s+1, d]
    merge_sort(a, s+1, d, tmp);

    // ucesljavamo segmente [l, s] i [s+1, d] smestajuci rezultat u
    // niz tmp
    merge(a, l, s, a, s+1, d, tmp, l);
    // vracamo rezultat iz niza tmp nazad u niz a
    for (int i = l; i <= d; i++)
      a[i] = tmp[i];

    // moze i pomocu biblioteckih funkcija
    /*
    merge(next(a.begin(), l), next(a.begin(), s+1),
      next(a.begin(), s+1), next(a.begin(), d+1),
      next(tmp.begin(), l));
    copy(next(tmp.begin(), l), next(tmp.begin(), d+1), next(a.begin(), l));
    */
  }
}

// sortira niz a
void merge_sort(vector<int>& a) {
  // alociramo pomocni niz
  vector<int> tmp(a.size());
  // pozivamo funkciju sortiranja
  merge_sort(a, 0, a.size() - 1, tmp);
}
```

Broj invezija u nizu

Problem: Odredi koliko različitih parova elemenata u nizu je takvo da je prvi element strogo veći drugog. Na primer, u nizu $5, 4, 3, 1, 2$ takvi su parovi $(5, 4)$, $(5, 3)$, $(5, 1)$, $(5, 2)$, $(4, 3)$, $(4, 1)$, $(4, 2)$, $(3, 1)$ i $(3, 2)$ i ima ih 9.

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);
}
```

Quick-sort

Problem: Sortirati niz brojeva primenom algoritma brzog sortiranja.

```
// soritra segment pozicija [l, d] u nizu a
void quick_sort(vector<int>& a, int l, int d) {
  // ako segment [l, d] jedan ili nula elementa on je vec sortiran
  if (l < d) {
    // za pivot uzimamo proizvoljan element segmenta
    swap(a[l], a[l + rand() % (d - l + 1)]);
    // particionisemo niz tako da se u njemu prvo javljaju elementi
    // manji ili jednaki pivotu, a zatim veci od pivota
    // tokom rada vazi [l, k] su manji ili jednaki pivotu
    // (k, i) su veci od pivota, [i, d] su jos neispitani
    int k = l;
```

```
    for (int i = l+1; i <= d; i++)
      if (a[i] <= a[l])
    swap(a[i], a[++k]);
    // razmenjujemo pivot sa poslednjim manjim ili jednakim elementom
    swap(a[l], a[k]);
    // rekurzivno sortiramo deo niza levo i desno od pivota
    quick_sort(a, l, k - 1);
    quick_sort(a, k + 1, d);
  }
}

// sortira niz a
void quick_sort(vector<int>& a) {
  // poziv pomocne funkcije koja u nizu a sortira segment pozicija [0, n-1]
  quick_sort(a, 0, a.size() - 1);
}
```

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 $O(n \log n)$.

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 (koja se nalazi u intervalu $[l, d]$, gde su l i d granice dela niza koji se obrađuje.

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 (po uzoru na QuickSort funkcija prima parametre l i d), međutim, pošto se u varijanti QuickSelect vrši samo jedan rekurzivni poziv i to kao repni, rekurziju je veoma jednostavno eliminisati.

Kao i u slučaju algoritma QuickSort i 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 $O(n)$. Zaista, ponašanje funkcije bismo mogli

opisati jendačinom $T(n) = T(n/2) + O(n)$, $T(0) = O(1)$, čije je rešenje $O(n)$. U prvom koraku se prilikom pivotiranja vrši n poređenja, u drugom $n/2$, u trećem $n/4$ i tako dalje, što se odozgo može ograničiti sa $2n$). Sa druge strane, ako bi pivot stalno bio blizak nekom od dva kraja intervala $[l, d]$, tada bi složenost algoritma bila $O(n^2)$. Zaista, ponašanje bi se tada moglo opisati pomoću $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$. U prvom koraku bismo imali n poređenja, u drugom $n-1$, u trećem $n-2$ što u zbiru daje $n(n+1)/2$. Sličnom analizom kao onom kojom smo pokazali da je prosečna složenost algoritma QuickSort jednaka $O(n \log n)$, može se pokazati da je prosečna složenost algoritma QuickSelect jednaka $O(n)$.

Pomenimo da izbor algoritma particionisanja može uticati na efikasnost, naročito u slučaju kada u nizu ima dosta ponovljenih elemenata. Algoritam koji particionisanje vrši tako što niz obilazi sa dva kraja i vrši razmene je u tom svetlu dosta efikasniji od algoritma koji niz obilazi samo sa jednog kraja.

```
// pronalazimo
int ktiElement(vector<int>& a, int l, int d, int k) {
  while (true) {
    // pivot dovodimo na poziciju l
    swap(a[l], a[random_value(l, d)]);
    // particionišemo elemente niza
    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--]);
    }
    // pivot vraćamo na poziciju j
    swap(a[l], a[j]);
    // pre pivota postoji bar k elemenata pa je dovoljno da
    // pretragu nastavimo samo u delu niza pre pivota
    if (k < j)
      d = j - 1;
    // zaključno sa pivoto
    else if (k > j)
      l = j + 1;
    // pivot je tačno k-ti po redu
    else
      return a[k];
  }
}

int ntiElement(vector<int>& a, int n) {
  return ntiElement(a, 0, a.size() - 1, n);
}
```

Na sličan način možemo odrediti i zbir najvećih k elemenata niza.

```
// QuickSelect - odredjujemo najvecih k elemenata niza a tj. niz permutujemo
// tako da se najvecih k elemenata nadju na prvih k pozicija (u proizvoljnom
// redosledu)
void qsortK(vector<int>& a, int l, int d, int k) {
  if (k <= 0 || l >= d)
    return;

  // niz particionisemo tako da se pivot (element a[l]) dovede na
  // svoje mesto, da ispred njega budu svi elementi koji su veci ili
  // jednaki od njega, a da iza njega budu svi elementi veci od njega
  int m = l;
  for (int t = l+1; t <= d; t++)
    if (a[t] >= a[l])
      swap(a[++m], a[t]);
  swap(a[m], a[l]);

  if (k < m - l)
    // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
    qsortK(a, l, m - 1, k);
  else
    // neki kod k najvecih su iza pivota - obradjujemo deo iza pivota
    qsortK(a, m+1, d, k - (m - l + 1));
}

// QuickSelect - pomocna funkcija zbog lepseg interfejsa
void qsortK(vector<int>& a, k) {
  qsortk(a, 0, a.size() - 1, k);
}

int zbirKNajvecih(vector<int>& a, int k) {
  // odredjujemo prvih k najvecih elemenata niza
  qsortK(a, k);

  // sabiramo prvih k elemenata niza i vraćamo rezultat
  int s = 0;
  for (int i = 0; i < k; i++)
    s += a[i];
  return s;
}
```

U jeziku C++ na raspolaganju imamo funkcije `nth_element` i `partial_sort` koje vrše delimično sortiranje niza. Funkcija `nth_element` organizuje niz tako da je n-ti element na svom mestu i da su svi elementi ispred njega manji (ne obavezno sortirani), dok `partial_sort` obezbeđuje da je sortirano prvih n elemenata (i da su oni manji od elemenata iza n-tog, koji ne moraju biti sortirani).

```
// niz particionisemo tako da je  n-ti element na svom mestu i da su
// svi elementi ispred njega manji ili jednaki od svih elemenata iza
```

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

// odredjujemo i ispisujemo zbir prvih k elemenata transformisanog niza
cout << accumulate(a.begin(), next(a.begin(), k), 0) << endl;
```

Minimum i maksimum niza

Problem: Pretpostavimo da se u nizu nalaze elementi koje je moguće upoređivati, ali čije je poređenje skupa operacija (na primer dugačke niske, koje se porede leksikografski). Definisati funkciju koja efikasno određuje minimum i maksimum tog niza.

Jasno je da složenost mora da bude $O(n)$, ali pošto je operacija poređenja skupa zanima nas i konstanta koja se javlja uz n tj. želimo da smanjimo broj poređenja.

Jedan način da se zadatak reši je da se minimum i maksimum nađu nezavisno i za to nam je potrebno $2n - 2$ operacija poređenja.

Možemo pokušati induktivno-rekurzivnim pristupom. Pretpostavimo da znamo minimum i maksimum svih elemenata niza osim poslednjeg. Maksimum i minimum ažuriramo tako što poslednji element sa tim minimumom i maksimumom i na taj način problem rešavamo sa 2 poređenja po članu niza, tj. opet sa oko $2n - 2$ poređenja (jer u baznom slučaju samo inicijalizujemo minimum i maksimum na početni član niza). Rekurentna jednačina u ovom slučaju je $T(n) = T(n-1) + 2$, $T(1) = 0$ i njeno je rešenje $2n - 2$. Recimo i da se pitanje da li je tekući element manji od minimuma može preskočiti ako je tekući element veći od maksimuma (jer element ne može istovremeno da bude i strogo veći od maksimuma i strogo manji od minimuma), ali to ne menja asimptotsku složenost najgoreg slučaja (koja u ovoj situaciji nastupa kod opadajućih nizova).

```
void minimax(const vector<string>& a, int& Min, int& Max) {
  Min = 0; Max = 0;
  for (int i = 1; i < a.size(); i++)
    if (a[i] > a[Max])
      Max = i;
    else if (a[i] < a[Min])
      Min = i;
}
```

Malo efikasnije rešenje se može dobiti dekompozicijom. Pretpostavimo da znamo minimum i maksimum dve polovine niza. Sa tačno dva poređenja možemo dobiti minimum i maksimum celog niza (minimum je manji od dva minimuma, a maksimum je veći od dva maksimuma).

```
void minimax(const vector<string>& a, int l, int d,
             int& Min, int& Max) {
  if (l == d) {
    Min = l; Max = l;
```

```
  } else if (l + 1 == d) {
    if (a[l] < a[d]) {
      Min = l;  Max = d;
    } else {
      Min = d;  Max = l;
    }
  } else {
    int s = l + (d-l) / 2;
    int MinL, MaxL, minD, maxD;
    minimax(a, l, s, MinL, MaxL);
    minimax(a, s+1, d, MinD, MaxD);
    Min = a[MinL] < a[MinD] ? minL : minD;
    Max = a[MaxL] < a[MaxD] ? maxD : maxL;
  }
}


void minimax(const vector<string>& a, int& Min, int& Max) {
   minimax(a, 0, a.size() - 1, Min, Max);
}
```

Pretpostavimo da je n stepen broja 2, tj. da postoji k tako da je $n = 2^k$. Ovo rešenje zadovoljava rekurentnu jednačinu $T(n) = 2T(n/2) + O(1)$, $T(1) = 0$, $T(2) = 1$ čije je rešenje, kao što znamo, $O(n)$, međutim, ovde nas zanima malo preciznija analiza tj. konstanta koja se javlja uz n. Važi:

$$T(2^k) = 2T(2^{k-1}) + 2$$

$$\frac{T(2^k)}{2^k} = \frac{T(2^{k-1})}{2^{k-1}} + \frac{1}{2^{k-1}}$$

Smenom $S(k) = \frac{T(2^k)}{2^k}$ dobijamo da je $S(k) = S(k-1) + \frac{1}{2^{k-1}}$ i sumiranjem dobijamo da je $S(k) = \frac{3}{2} - 2^{1-k}$, pa je $T(n) = \frac{3}{2}n - 2$.

Ovo rešenje je stoga efikasnije od prethodnih (mada je asimptotska složenost svih rešenja $O(n)$). Empirijska analiza testiranjem na test primerima pokazuje da nema značajne razlike u vremenu izvršavanja

Maksimalni zbir segmenta

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

Na kraju pokažimo još jedan način da rešimo ovaj problem, zasnovan na tehnici razlaganja.

Dekompozicija nam sugeriše da je poželjno da niz podelimo na dva podniza jednake dužine čija rešenja možemo da konstruišemo na osnovu induktivne hipoteze (najčešće rekurzivnim pozivima). Bazu i ovaj put 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.

```
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});
}

int maksZbirSegmenta(int a[], int n) {
  return maksZbirSegmenta(a, 0, n - 1);
}
```

Analiza složenosti ovaj put zahteva kompleksniji matematički aparat, ali je prilično jednostavna kada se taj aparat poznaje. Naime, 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 $T(n) = 2T(n/2) + O(n)$. Naime, vrše se dva rekurzivna poziva za duplo manje nizove, a najveći zbir segmenata koji obuhvataju središnji element izračunavamo u vremenu $O(n)$ (što je prilično očigledno jer imamo dve petlje koje se ukupno izvršavaju n puta, a čija su tela konstantne složenosti). Na

osnovu master teoreme lako se zaključuje da je $T(n) = O(n \log n)$. Dakle, ovaj algoritam je manje efikasan od prethodna dva, ali je i dalje prilično upotrebljiv, jer je mnogo bolji od početna dva veoma naivna pokušaja.

Postoje i drugi algoritmi složenosti $O(n)$ za rešavanje ovog problema (na primer, onaj dobijen odsecanjima u pretrazi, Kadanov algoritam ili agoritam dobijen preko prefiksnih zboriva), međutim na nizovima manjim od milion elemenata se ne može primetiti razlika između njih i između prethodnog algoritma složenosti $O(n \log n)$. Fine razlike između ovih algoritama se vide tek na nizu od 10 miliona elemenata i tada sve tri implementacije složenosti $O(n)$ posao završavaju za oko $0,120$ sekundi, a ona zasnovana na dekompoziciji za oko $0,330$ sekundi.

Ipak, na ideji dekompozicije možemo izgraditi i efikasniji algoritam. Ključni uvid je da se najveći zbir segmenta oko srednjeg elementa 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, ali moramo vratiti dug 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.

```
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, 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, 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;
}
```

Složenost prethodne implementacije zadovoljava jednačinu $T(n) = 2T(n/2) + O(1)$ i jednaka je $O(n)$.

Silueta zgrada - skyline

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) {
  }
};
```

Silueta je deo-po-deo konstantna funkcija i određena je intervalima konstantnosti $(-\infty, x_0)$, $[x_0, x_1)$, $[x_1, x_2)$, ..., $[x_{n-1}, +\infty)$, 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.

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, koja se trivijalno određuje). 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. Ključna opaska je to da 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;

  // silueta koja odgovara jednoj zgradi
  if (l == d) {
    rezultat.emplace_back(zgrade[l].a, zgrade[l].h);
    rezultat.emplace_back(zgrade[l].b, 0);
    return rezultat;
  }

  // određujemo posebno siluete za prvu i drugu polovinu zgrada
  int s = l + (d - l) / 2;
  vector<promena> rezultat_l = silueta(zgrade, l, s);
  vector<promena> rezultat_d = silueta(zgrade, s+1, d);

  // objedinjujemo dve siluete

  // tekući indeksi i visine u levoj i desnoj silueti
  int ll = 0, dd = 0;
  int Hl = 0, Hd = 0;
  // dok god postoji neka neobrađena promena
  while (ll < rezultat_l.size() || dd < rezultat_d.size()) {
    // određujemo novu tačku promene
    int x;
    // ako smo završili sa levom siluetom samo prebacujemo zgrade iz desne
    if (ll == rezultat_l.size()) {
      x = rezultat_d[dd].x; Hd = rezultat_d[dd].h;
      dd++;
    // ako smo završili sa desnom siluetom samo prebacujemo zgrade iz desne
    } else if (dd == rezultat_d.size()) {
      x = rezultat_l[ll].x; Hl = rezultat_l[ll].h;
      ll++;
    } else {
      // određujemo raniju od tekućih promena leve i desne siluete
      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++;
      }
    }

    // veća od dve tekuće visine
    int h = max(Hl, Hd);

    // integrišemo (x, h) u tekuću rezultujući siluetu
    if (rezultat.size() > 0) {
      // poslednja promena tekuće siluete
      int xb = rezultat.back().x, hb = rezultat.back().h;
      // ako se promena dešava na istoj x koordinati kao prethodna
      // samo menjamo visinu
      if (x == xb)
        rezultat.back().h = h;
      // ako nova promena ima istu visinu kao prethodna, preskačemo je
      else if (h != hb)
        // u suprotnom dodajemo novu promenu u niz
        rezultat.emplace_back(x, h);
    } else
      // nema prethodne promene, pa novu promenu dodajemo na početak niza
      rezultat.emplace_back(x, h);
  }

  return rezultat;
}

vector<promena> silueta(const vector<zgrada>& zgrade) {
   return silueta(zgrade, 0, zgrade.size() - 1);
}
```

Karacubin algoritam množenja polinoma

Problem: Definisati funkciju koja množi dva polinoma predstavljena vektorima svojih koeficijenata. Jednostavnosti radi pretpostaviti da su vektori dužine 2^k.

Klasičan algoritam množenja koji se uči u srednjoj školi ima složenost $O(n^2)$.

```
// funkcija mnozi dva polinoma p1*p2
vector<double> proizvod(const vector<double>& p1,
                        const vector<double>& p2) {
  vector<double> proizvod(2*n-1, 0);
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      proizvod[i+j] += p1[i] * p2[j];
  return c;
}
```

Iako se neko vreme smatralo da je to donja granica složenosti, 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, što nije kritično, jer sa sabiranje i oduzimanje obično vrši brže nego množenje, a što važi i za polinome, jer je sabiranje i oduzimanje polinoma operacija linearne složenosti). 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 (oni su potrebni i direktno i za izračunavanje proizvoda $ad+bc$).

Imajući ovaj Karacubin "trik" u vidu, lako možemo napraviti algoritam zasnovan na dekompoziciji.

```
// funkcija mnozi dva polinoma p1*p2 sa po 2^k koeficijenata
vector<double> karacuba(const vector<double>& p1,
                        const vector<double>& p2) {
  // broj koeficijenata polinoma
  int n = p1.size();

  // polinome stepena 0 direktno množimo
  if (n == 1)
    return vector<double(1, p1[0] * p2[0])

  // delimo p1 na dve polovine: a i b
  vector<double> a(n / 2), b(n / 2);
  copy_n(begin(p1), n/2, begin(a));
  copy_n(next(begin(p1), n/2), n/2, begin(b));

  // delimo p2 na dve polovine: c i d
  vector<double> c(n / 2), d(n / 2);
  copy_n(begin(p2), n/2, begin(c));
  copy_n(next(begin(p2), n/2), n/2, begin(d));

  // Važi:
  //   (ax+b)*(cx+d) = a*c*x^2 + ((a+b)*(c+d) - a*c - b*d)*x + b*d

  // rekurzivno računamo a*c i b*d
  vector<double> ac = karacuba(a, c);
  vector<double> bd = karacuba(b, d);

  // izračunavamo a+b (koristimo pomoćni vektor a)
  for (int i = 0; i < n/2; i++)
    a[i] += b[i];
  // izračunavamo c+d (koristimo pomoćni vektor b)
  for (int i = 0; i < n/2; i++)
    c[i] += d[i];

  // izračunavamo (a+b)*(c+d)
```

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

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

  // vraćamo rezultat
  return proizvod;
}
```

Složenost prethodnog 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 testovi pokazuju da 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. Pažljivija analiza pokazuje da je moguće svu pomoćnu memoriju alocirati samo jednom i onda tokom rekurzije koristiti stalno isti pomoćni memorijski prostor. Veličina potrebne pomoćne memorije je $4n$ (dva puta po n da se smeste polinomi $a + b$ i $c + d$ i još $2n$ da se smesti njihov proizvod). Dodatna optimizacija je da se primeti da je za male stepene polinoma klasičan algoritam brži nego algoritam zasnovan na dekompoziciji (ovo je čest slučaj kod algoritama zasnovanih na dekompoziciji). Eksperimentalnom analizom se utvrđuje da se više isplati primeniti klasičan algoritam kad god je $n \leq 4$.

```
// množimo polinome čiji su koeficijenti smešteni u vektorima
// p1[start1, start1+n) i p2[start2, start2+n)
// i rezultat smeštamo u vektor
// proizvod[start_proizvod, start_proizvod + 2n),
// koristeći pomocni memorijski prostor u vektoru
// pom[start_pom, start_pom + 4n)
void karacuba(int n,
              const vector<double>& p1, int start1,
              const vector<double>& p2, int start2,
              vector<double>& proizvod, int start_proizvod,
              vector<double>& pom, int start_pom) {

  // izlaz iz rekurzije
  if (n <= 4) {
    // klasični algoritam množenja
    for (int i = 0; i < 2*n; i++)
```

```
      proizvod[start_proizvod + i] = 0;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        proizvod[start_proizvod + i+j] +=
           p1[start1 + i] * p2[start2 + j];
    return;
  }

  // Važi: (a+bx)*(c+dx) =
  //           a*c + ((a+b)*(c+d) - a*c - b*d)*x + b*d*x^2

  // Izračunavamo rekurzivno a*c i smeštamo ga u levu polovinu
  // proizvoda
  karacuba(n / 2, p1, start1, p2, start2,
           proizvod, start_proizvod, pom, start_pom);
  // Izračunavamo rekurzivno b*d i smeštamo ga u desnu polovinu
  // proizvoda
  karacuba(n / 2, p1, start1 + n/2, p2, start2 + n/2,
           proizvod, start_proizvod + n, pom, start_pom);

  // Izračunavamo a+b i smeštamo ga u pomoćni vektor (na početak)
  for (int i = 0; i < n/2; i++)
    pom[start_pom + i] =
        p1[start1 + i] + p1[start1 + n/2 + i];
  // Izračunavamo c+d i smeštamo ga u pomoćni vektor (iza (a+b))
  for (int i = 0; i < n/2; i++)
    pom[start_pom + n / 2 + i] =
        p2[start2 + i] + p2[start2 + n/2 + i];

  // Rekurzivno izračunavamo (a+b)*(c+d) i smeštamo ga
  // u pomoćni vektor, iza (a+b) i (c+d)
  karacuba(n / 2, pom, start_pom, pom, start_pom + n / 2,
           pom, start_pom + n, pom, start_pom + 2*n);

  // Izračunavamo (a+b)*(c+d) - (ac + bd)
  for (int i = 0; i < n; i++)
    pom[start_pom + n + i] -=
      proizvod[start_proizvod + i] + proizvod[start_proizvod + n + i];

  // Dodajemo ad+bc na sredinu proizvoda
  for (int i = 0; i < n; i++)
    proizvod[start_proizvod + n/2 + i] += pom[start_pom + n + i];
}

// funkcija množi dva polinoma p1*p2 sa po 2^k koeficijenata
vector<double> karacuba(const vector<double>& p1,
                        const vector<double> p2) {
  int n = p1.size();
  // koeficijenti proizvoda
  vector<double> proizvod(2 * n);
```

```
  // pomoćni memorijski prostor potreban za realizaciju algoritma
  vector<double> pom(4 * n);
  // vršimo množenje
  karacuba(n, p1, 0, p2, 0, proizvod, 0, pom, 0);
  // vraćamo proizvod
  return proizvod;
}
```

FFT - brza Furijeova transformacija

Još brži način množenja dva polinoma zasnovan je na čuvenoj *brzoj Furijeovoj transformaciji*, koja je korisna i u mnogim drugim kontekstima. Lep opis ove transformacije dostupan je u knjizi profesora Miodraga Živkovića.

Razmotrimo ovde pitanje implementacije ovog algoritma. U jeziku C++ kompleksne brojeve imamo na raspolaganju u obliku tipova `complex<double>` i `complex<float>` (zapis u dvotrukoj i jednostrukoj tačnosti). Direktan način implementacije je sledeći.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// brza Furijeova transformacija vektora a duzine n=2^k
// bool parametar inverzna odredjuje da li je direktna ili inverzna
ComplexVector fft(const ComplexVector& a, bool inverzna) {
  // broj koeficijenata polinoma
  int n = a.size();

  // ako je stepen polinoma 0, vrednost u svakoj tacki jednaka
  // je jedinom koeficijentu
  if (n == 1)
    return ComplexVector(1, a[0]);

  // izdvajamo koeficijente na parnim i na neparnim pozicijama
  ComplexVector A(n / 2), B(n / 2);
  for (int i = 0; i < n / 2; i++) {
    A[i] = a[2 * i];
    B[i] = a[2 * i + 1];
  }

  // rekurzivno izracunavamo Furijeove transformacije tih polinoma
  ComplexVector fftA = fft(A, inverzna),
                fftB = fft(B, inverzna);

  // objedinjujemo rezultat
  ComplexVector rez(n);
  for (int k = 0; k < n; k++) {
    // odredjujemo primitivni n-ti koren iz jedinice
```

```
    double coeff = inverzna ? -1.0 : 1.0;
    complex<double> w = exp((coeff * 2 * k * M_PI / n) * 1i);
    // racunamo vrednost polinoma u toj tacki
    rez[k] = fftA[k % (n / 2)] + w * fftB[k % (n / 2)];
  }
  // vracamo konacan rezultat
  return rez;
}


// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  return fft(a, false);
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  ComplexVector rez = fft(a, true);
  // nakon izracunavanja vrednosti, potrebno je jos podeliti
  // sve koeficijente duzinom vektora
  int n = a.size();
  for (int k = 0; k < n; k++)
    rez[k] /= n;
  return rez;
}
```

Jasno je da prethodna procedura zadovoljava jednačinu $T(n) = 2T(n/2)+O(n)$, pa joj je složenost $O(n \log n)$. Nakon transformacije (promene reprezentacije izračunavanjem vrednosti) dva polinoma njihovo množenje je moguće u vremenu $O(n)$, pa je ukupna složenost množenja polinoma pomoću FFT jednaka $O(n \log n)$.

Međutim, prethodnu naivnu implementaciju je moguće poboljšati. Najveći problem je to što se primitivni n-ti koreni iz jedinice računaju zasebno u svakom rekurzivnom pozivu. Efikasnost bi se značajno popravila ako bi se niz korena izračunao samo jednom, pre funkcije. Problem je u tome što se tokom rekurzije n smanjuje, pa su nam u svakom pozivu potrebni različiti koreni. Međutim, ako je neki broj k-ti primitivni koren iz jedinice, onda je on i $2k$-ti koren primitivni koren iz jedinice. Zato, ako znamo niz primitivnih n-tih korena iz jedinice ($e^{\frac{2k\pi i}{n}}$, za k od 0 do $n-1$), tada su elementi na parnim pozicijama tog vektora $n/2$-ti primitivni koreni iz jedinice. Zaista, ako je $k = 2k'$, tada je $e^{\frac{2k\pi i}{n}} = e^{\frac{2k'\pi i}{\frac{n}{2}}}$ i važi da ako je $0 \leq k < n$, tada je $0 \leq k' < n/2$. Dakle, u početku možemo izračunati niz svih primitivnih n-tih korena iz jedinice i njih koristiti na početnom nivou rekurzije, na narednom nivou rekurzije ćemo koristiti svaki drugi element tog vektora, na narednom svaki četvrti, zatim svaki osmi i tako dalje. Na primer, ako je $n = 8$, početni niz korena je 1, $\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$, i, $-\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$, -1, $-\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}$,

$-i$, $\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}$, dok je na narednom nivou rekurzije $n = 4$ i koriste se koreni 1, i, -1, $-i$, dok je na narednom nivou rekurzije $i = 2$ i koriste se 1 i -1.

Još jedan problem prethodne implementacije je to što se tokom rekurzije alociraju i popunjavaju pomoćni vektori, što dovodi do gubitka i vremena i memorije. Furijeovu transformaciju je moguće realizovati i bez korišćenja pomoćne memorije. Na početnom nivou rekurzije koeficijenti ulaznog polinoma su svi dati počenim vektorima koeficijenata. Na narednom se posmatraju elementi na pozicijama 0, 2, 4, ..., $n-2$ i na pozicijama 1, 3, ..., $n-1$. Na narednom se posmatraju elementi na pozicijama 0, 4, 8, $n-4$, zatim elementi na pozicijama 1, 5, ..., $n-3$, zatim elemeni na pozicijama 2, 6, ..., $n-2$, i na kraju elementi na pozicijama 3, 7, ..., $n-1$. Slično se nastavlja i na daljim nivoima rekurzije. Dakle, umesto formiranja pomoćnog ulaznog vektora sa pogodno odabranim ulaznim koeficijentima, prosleđivaćemo originalni vektor, poziciju početka s i pomeraj d i posmatraćemo njegove elemente na pozicijama $s + dk$, za $0 \leq k < n$, gde je $n = n_0/d$, a n_0 je dužina početnog vektora. Rezultate rekurzivnih poziva možemo smestiti u dve polovine rezultujućeg niza. Nakon toga rezultate objedinjavamo. Vrednosti na pozicijama k i $k + n/2$ rezultujućeg vektora određene su vrednostima na poziciji k u rezultatu prvog i drugog rekurzivnog, međutim, one se nalaze upravo na pozicijama k i $k + n/2$ rezultujućeg vektora (jer smo rezultate rekurzivnih poziva smestili u prvu i drugu polovinu rezultata). Moramo voditi računa da te dve vrednosti moramo istovremeno izračunati i ažurirati.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// funkcija vrši Furijeovu transformaciju (direkntu ili inverznu) elemenata
//   a[start_a], a[start_a + step], a[start_a + 2step], ...
// i rezultat smešta u niz
//   rez[start_rez], rez[start_rez + 1], rez[start_rez + 2], ...
// koristeci primitivne korene iz jedinice smestene u niz
//   w[0], w[step], w[2step], ...
void fft(const ComplexVector& a, int start_a,
         const ComplexVector& w,
         ComplexVector& rez, int start_rez,
         int step) {
  // broj elemenata niza koji se transformise
  int n = a.size() / step;

  // stepen polinoma je nula, pa mu je vrednost u svakoj tacki jednaka
  // konstantnom koeficijentu
  if (n == 1) {
    rez[start_rez] = a[start_a];
    return;
  }

  // rekurzivno transformisemo niz koeficijenata na parnim pozicijama
  // smestajuci rezultat u prvu polovinu niza rez
  fft(a, start_a, w, rez, start_rez, step*2);
```

```
  // rekurzivno transformisemo niz koeficijenata na neparnim pozicijama
  // smestajuci rezultat u drugu polovinu niza rez
  fft(a, start_a + step, w, rez, start_rez + n/2, step*2);

  // objedinjujemo dve polovine u rezultujuci niz
  for (int i = 0; i < n/2; i++) {
    auto r1 = rez[start_rez + i];
    auto r2 = rez[start_rez + (i + n/2)];
    rez[start_rez + i] = r1 + w[i*step] * r2;
    rez[start_rez + (i + n/2)] = r1 - w[i*step] * r2;
  }
}

// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n/2);
  for (int k = 0; k < n/2; k++)
    w[k] = exp((2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector rez(n);
  // vrsimo transformaciju
  fft(a, 0, w, rez, 0, 1);
  // vracamo dobijeni rezultat
  return rez;
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n);
  for (int k = 0; k < n; k++)
    w[k] = exp((- 2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector rez(n);
  // vrsimo transformaciju
  fft(a, 0, w, rez, 0, 1);
  // popravljamo rezultat
  for (int i = 0; i < n; i++)
    rez[i] /= n;
  // vracamo dobijeni rezultat
  return rez;
}
```

```
vector<double> proizvod(const vector<double>& p1,
                        const vector<double>& p2) {
  // duzina niza koeficijenata
  int n = p1.size();
  // kreiramo nizove kompleksnih koeficijenata dopunjavajuci ih nulama
  // do dvostruke duzine
  int N = 2*n;
  ComplexVector a(N, 0.0);
  copy(begin(p1), end(p1), begin(a));
  ComplexVector b(N, 0.0);
  copy(begin(p2), end(p2), begin(b));

  // vrsimo Furijeove transformacije oba vektora koeficijenata
  ComplexVector va = fft(a), vb = fft(b);
  // prolazimo Furijeovu transformaciju vektora koeficijenata proizvoda
  ComplexVector vc(N);
  for (int i = 0; i < N; i++)
    vc[i] = va[i] * vb[i];
  // inverznom Furijeovom transformacijom rekonstruisemo koeficijente
  // proizvoda
  ComplexVector c = ifft(vc);

  // realne delove kompleksnih brojeva smestamo u niz rez i vracamo ga
  vector<double> rez(N);
  transform(begin(c), end(c), begin(rez),
            [](Complex x){
              return real(x);
            });
  return rez;
}
```

Pažljivom analizom je moguće ukloniti rekurziju iz prethodne implementacije (tako se dobija Kuli-Tukijev nerekurzivni algoritam za FFT, zasnovan na tzv. leptir-shemi), no time se nećemo baviti u sklopu ovog kursa.

Štrasenov algoritam množenja matrica

Problem: Napisati program koji određuje proizvod dve date matrice.

Naivni algoritam za množenje matrica je složenosti $O(n^3)$.

```
typedef vector<vector<int>> Matrica;

void alociraj(Matrica& m, int v, int k) {
  m.resize(v);
  for (int i = 0; i < v; i++)
     m[i].resize(k, 0);
}
```

```
Matrica proizvod(Matrica a, Matrica b) {
  Matrica c;
  c.alociraj(v, a.size(), b[0].size());
  for (int i = 0; i < n; i++)
     for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
           c[i][j] += a[i][k] * b[k][j];
  return c;
}
```

Postavlja se pitanje da li je efikasniji algoritam moguće izvesti dekompozicijjom. Ako pretpostavimo da je dimenzija matrice stepen broja 2, tada se oba činioca mogu predstaviti putem četiri podmatrice čije su dimenzije $n/2$. Potrebno je, dakle, izračunati proizvod

$$\left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \cdot \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

n Direktan pristup zahteva izračunavanje proizvoda

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Pošto sabiranje matrica zahteva $O(n^2)$ operacija, ovo rešenje zadovoljava rekurentnu jednačinu $T(n) = 8T(n/2) + O(n^2)$, čije je rešenje, znamo na osnovu master teoreme, jednako $O(n^{\log_2 8}) = O(n^3)$.

Slično kao kod Karacube, ali dosta komplikovanije, Štrasen je primetio da je moguće upotrebiti samo 7 množenja manjih blok matrica. Koristimo 7 pomoćnih matrica.

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned}$$

I pomoću njih dobijamo 4 tražene matrice.

$$
\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Naravno, prethodne formule nije neophodno učiti napamet.

Time dobijamo algoritam koji zadovoljava jednačinu $T(n) = 7T(n/2) + O(n^2)$, čije je rešenje, znamo na osnovu master teoreme jednako $O(n^{\log_2 7}) \approx O(n^{2,8074})$.

Ovaj dobitak na vremenu ima svoju cenu i Štrasenov algoritam ima nekoliko ozbiljnih nedostataka.

- Štrasenov algoritam je komplikovaniji i teži za implementaciju od običnog.
- Kako je to obično slučaj, dekompozicija postaje korisna tek za probleme većih dimenzija. Praktično se pokazuje da je potrebno da dimenzija bude veća od 100, da bi se osetio dobitak. Ova činjenica se koristi tako što se kao baza indukcije tj. izlaz iz rekurzije upotrebi slučaj kada je n manje od oko stotine i tada se primeni naivni algoritam.
- Troši se dodatna memorija, što za početno proširenje matrica tako da im dimenzija bude stepen dvojke, što na smeštanje pomoćnih matrica.
- Štrasenov algoritam je manje numerički stabilan od običnog. Za iste veličine greške ulaznih podataka, Štrasenov algoritam obično dovodi do većih grešaka u izlaznim podacima.
- Štrasenov algoritam se teže paralelizuje.

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 $O(n^2)$.

```
struct Tacka {
  int x, y;
};

double rastojanje(const Tacka& t1, const Tacka& t2) {
  double dx = t1.x - t2.x;
  double dy = t1.y - t2.y;
  return sqrt(dx*dx + dy*dy);
}
```

```
double najbliziParTacaka(const vector<Tacka>& t) {
  double d = numeric_limits<double>::max();
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++) {
      double dij = rastojanje(tacke[i], tacke[j]);
      if (dij < d)
        d = dij;
    }
  return d;
}
```

Jedan način da se do rešenja dođe efikasnije je da se primeni dekompozicija.

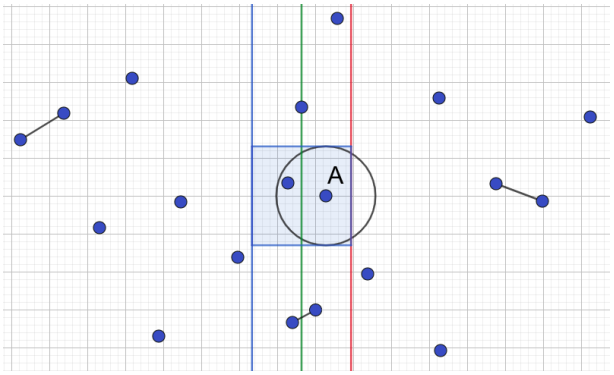
Bazni slučaj predstavlja situacija u kojoj imamo manje od četiri tačke, jer njih ne možemo podeliti u dve polovine u kojima postoji bar po jedan par tačaka (a ako u skupu nemamo bar 2 tačke, najmanje rastojanje nije jasno definisano). U tom slučaju rešenje nalazimo poređenjem rastojanja svih parova tačaka (pošto je tačaka malo, ovaj korak je složenosti $O(1)$).

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 (moguće je da se svih n tačaka nađe u tom pojasu) i ako ispitujemo sve parove, dolazimo u najgorem slučaju do oko $n^2/4$ poređenja (ako je pola tačaka levo, a pola desno od linije podele). Ipak, proveru je moguće organizovati tako da se proveri samo mali broj parova tačaka.

Jednostavnosti radi ćemo pretpostaviti da istovremeno razmatramo sve tačke unutar pojasa $[x - d, x + d]$, bez obzira sa koje strane vertikalne linije se nalaze (unapred znamo da je provera tačaka koje su sa iste strane vertikalne linije podele nepotrebna, ali ne može narušiti korektnost, dok god smo sigurni da se porede i svi potrebni parovi tačaka sa različite strane te linije). 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 $[y_A - d, y_A + d]$.



Slika 6: Najbliži par tačaka u levom pojasu, desnom pojasu i između pojaseva. Krug i kvadrat kojim su određeni kandidati oko tačke A.

Dalje smanjenje broja poređenja možemo dobiti ako primetimo da svaki par obrađujemo dva puta (jednom dok obrađujemo tačke u okolini prve, a jednom dok obrađujemo tačke u okolini druge tačke). 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 $[y_A, y_A + d]$. Prvi uslov možemo obezbediti tako što pre poređenja sve tačke iz pojasa širine d oko vertikalne linije podele izdvojimo u poseban niz (za to nam je potrebno $O(n)$ dodatne memorije i vremena). Drugi uslov efikasnije možemo obezbediti ako sve tačke tog pomoćnog niza sortiramo po koordinati y (za to nam je potrebno vreme $O(n \log n)$ 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 $y_A + d$ (ona od tačke A ne može biti na manjem rastojanju od d).

Na osnovu prethodne diskusije možemo napraviti sledeću implementaciju.

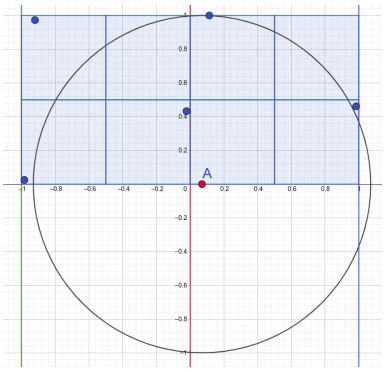
```
bool porediX(const Tacka& t1, const Tacka& t2) {
  return t1.x <= t2.x;
}

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

```
// funkcija određuje najbliži par tačaka u delu niza [l, r]
double najblizeTacke(vector<Tacka>& tacke, int l, int r) {
  // ako ima manje od 4 tačke, najmanje rastojanje pronalazimo
  // grubom silom
  if (r - l + 1 < 4) {
    // poredimo sve parove tačaka, tražeći minimalno rastojanje
    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;
  }

  // pozicija središnje tačke
  int s = l + (r - l) / 2;
  // određujemo najmanje rastojanje tačaka koje su levo od
  // središnje tačke (uključujući i nju)
  double d1 = najblizeTacke(tacke, l, s);
  // određujemo najmanje rastojanje tačaka koje su desno od
  // središnje tačke
  double d2 = najblizeTacke(tacke, s+1, r);
  // određujemo manje od ta dva rastojanja
  double d = min(d1, d2);
  // granice pojasa u kom se nalaze tačke za koje u suprotnom
  // pojasu mogu postajati tačke koje su im bliže od d
  double dl = tacke[s].x - d, dr = tacke[s].x + d;
  // određujemo sve tačke u tom pojasu
  vector<Tacka> pojas;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas.push_back(tacke[i]);
  // sortiramo ih na osnovu koordinate y
  sort(begin(pojas), end(pojas), porediY);
  // obrađujemo sve tačke iz pojasa u rastućem redosledu
  // koordinata y
  for (int i = 0; i < pojas.size(); i++)
    // svaku tačku poredimo sa tackama iznad nje koje su u
    // pravougaoniku visine d
    for (int j = i+1; j < pojas.size() && pojas[j].y - pojas[i].y < d; j++) {
      // ako su dve tačke na rastojanju manjem od d, ažuriramo d
      double dij = rastojanje(pojas[i], pojas[j]);
      if (dij < d)
        d = dij;
    }

  // vraćamo pronađeno minimalno rastojanje
  return d;
```

```
}

// funkcija koja određuje rastojanje između dve najbliže tačke
// u datom skupu tačaka
double najblizeTacke(vector<Tacka>& tacke) {
  // sortiramo tačke na osnovu koordinate x
  sort(begin(tacke), end(tacke), porediX);
  // određujemo najbliži par tačaka rekukrzivnom funkcijom
  return najblizeTacke(tacke, 0, tacke.size() - 1);
}
```

Odredimo složenost prethodne funkcije. Algoritam se sastoji od dva rekurzivna poziva za dvostruko manju dimenziju niza tačaka i faze dobijanja krajnjeg rezultata na osnovu rezultata rekurzivnih poziva i dodatne analize tačaka u pojasu $[x - d, x + d]$. Već smo konstatovali da izdvajanje tačaka centralnog pojasa zahteva $O(n)$ memorije i vremena i da sortiranje tih tačaka po koordinati y zahteva dodatnih $O(n \log n)$ koraka. Ostaje još da se proceni složenost ugnežđenih petlji u kojima se porede tačke unutar pojasa. Iako deluje da je složenost kvadratna, elementarnim geometrijskim rezonovanjem dokazaćemo da je složenost tog koraka linearna tj. $O(n)$ i da se u svakom koraku spoljašnje petlje unutrašnja petlja može izvršiti samo veoma mali broj puta (dokazaćemo da je taj broj izvršavanja ograničen odozgo sa 7, mada je u praksi on često i dosta manji od toga i za nasumično generisane tačke ta petlja se najčešće izvršava 0, 1 ili eventualno dva puta).

Za svaku tačku A možemo konstruisati 8 kvadrata dimenzije $d/2$, kao što je prikazano na slici (kvadrati su upisani u pojas $[x - d, x + d]$, u dva reda od po četiri kvadrata i tačka A leži na donjoj ivici donjih kvadrata).



Slika 7: Najbliži par tačaka

Najveće rastojanje između dve tačke unutar nekog kvadrata se postiže kada oni leže u njegovim naspramnim temenima, a pošto je dužina dijagonale kvadrata stranice $\frac{d}{2}$ jednaka $\frac{d\sqrt{2}}{2} \approx 0,70711 \cdot d$, rastojanje između svake dve tačke unutar istog kvadrata je strogo manje od d. Pošto svi kvadrati leže bilo potpuno sa

leve strane vertikalne linije podele, bilo sa njene desne strane unutar svakog od kvadrata se može naći najviše jedna tačka našeg skupa (u suprotnom bi se bilo sa leve, bilo sa desne strane centralne linije podele nalazio par tačaka sa rastojanjem strogo manjim od d, što je kontradiktorno sa definicijom veličine d). To znači da se iznad tačke A može nalaziti najviše 7 tačaka koje pripadaju ostalim kvadratima (sama tačka A već pripada jednom od kvadrata) i da se sve ostale tačke koje su iznad A nalaze i iznad naših kvadrata, što znači da im je rastojanje od A sigurno veće od d (jer im je vertikalno rastojanje veće od d) i njih nije potrebno razmatrati.

Tačke koje su sa iste strane linije podele kao i tačka A možemo prosto preskočiti u telu unutrašnje petlje i tako uštediti na računanju njihovog rastojanja od tačke A, ali eksperimenti pokazuju da ta ušteda nije osetna. Druga mogućnost za implementaciju je da ne čuvamo sve tačke iz pojasa u istom skupu, već da ih podelimo u dva pojasa i da zatim da obradimo prvo sve tačke iz levog pojasa gledajući rastojanja u odnosu na naredne najviše 4 tačke iz desnog pojasa, a zatim da obradimo sve tačke iz desnog pojasa gledajući rastojanja u odnosu na najviše 4 tačke iz levog pojasa (jer u suprotnom pojasu postoji 4 kvadrata dimezije $d/2$, za koje smo dokazali da ne mogu da sadrže dve tačke istovremeno). Implementacija na taj način je malo komplikovanija, a eksperimenti ne ukazuju na značajne dobitke.

Dakle, nakon rekurzivnih poziva, za dobijanje konačnog rezultata je potrebno izvršiti dodatnih $O(n \log n)$ koraka i dekompozicija zadovoljava rekurentnu jednačinu $T(n) = 2T(n/2) + O(n \log n)$. Rešenje ove jednačine, na osnovu master teoreme, je $O(n(\log n)^2)$.

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 (uobičajenim algoritmom objedinjavanja, zasnovanom na tehnici dva pokazivača, koji je dostupan i pomoću bibliotečke funkcije `merge` i posao obavlja u linearnoj složenosti). Na taj način dobijamo algoritam koji zadovoljava jednačinu $T(n) = 2T(n/2) + O(n)$ i složenosti je $O(n \log n)$. Naglasimo da ova optimizacija nije revolucionarna, ali može malo poboljšati efikasnost.

Na nivou implementacije, malo poboljšanje bismo mogli dobiti i tako što bismo izbegli alokacije pomoćnog vektora unutar rekurzivnih poziva i kod izmeniti tako da se u svakom rekurzivnom pozivu koristi isti, unapred alociran pomoćni vektor. Još jedna moguća optimizacija o kojoj bi se moglo razmisliti je smanjivanje broja operacija korenovanja.

```
// funkcija određuje najbliži par tačaka u delu niza [l, r]
// i sortira taj deo niza po koordinati y
double najblizeTacke(vector<Tacka>& tacke, int l, int r,
                     vector<Tacka>& pom) {
```

```
// ako ima manje od 4 tačke, najmanje rastojanje pronalazimo
// grubom silom
if (r - l + 1 < 4) {
  // poredimo sve parove tačaka, tražeći minimalno rastojanje
  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;
    }
  // sortiramo niz po koordinati y
  sort(next(begin(tacke), l), next(begin(tacke), r+1), porediY);
  return d;
}

// pozicija središnje tačke
int s = l + (r - l) / 2;
// određujemo najmanje rastojanje tačaka koje su levo od
// središnje tačke (uključujući i nju)
// određujemo najmanje rastojanje tačaka koje su desno od
// središnje tačke
double d2 = najblizeTacke(tacke, s+1, r);
// određujemo manje od ta dva rastojanja
double d = min(d1, d2);

// objedinjavamo dva sortirana niza tačka bibliotečkom funkcijom
// u novi sortirani niz pom
merge(next(begin(tacke), l), next(begin(tacke), s+1),
      next(begin(tacke), s+1), next(begin(tacke), r+1),
      begin(pom), porediY);
// vraćamo sortirane tačke u originalni niz
copy(begin(pom), end(pom), next(begin(tacke), l));

// izdvajamo tačke iz pojasa [x-d, x+d] u vektor pom
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];

// svaku tačku poredimo sa tačkama iznad nje koje su u
// pravougaoniku visine d
for (int j = i+1; j < pojas.size() && pojas[j].y - pojas[i].y < d; j++) {
  // ako su dve tačke na rastojanju manjem od d, ažuriramo d
  double dij = rastojanje(pojas[i], pojas[j]);
  if (dij < d)
    d = dij;
}
```

```
  return d;
}

// funkcija određuje najbliži par datih tačaka, sortirajući usput
// niz po koordinati y (to je samo sporedni efekat nastao u cilju
// efikasne implementacije)
double najblizeTacke(vector<Tacka>& tacke) {
  vector<Tacka> pom(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pom);
}
```

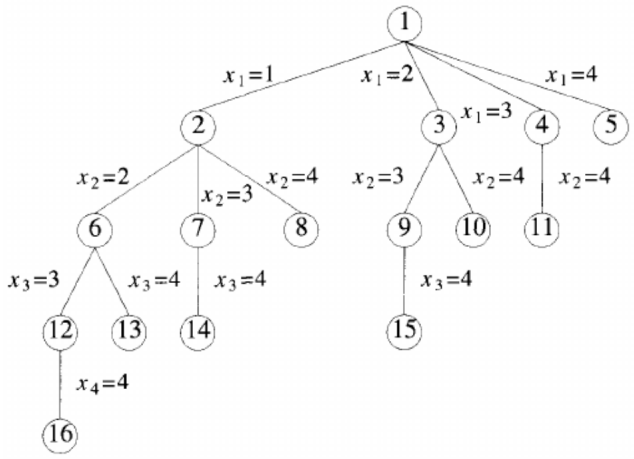
Čas 10.1, 10.2, 10.3 - Pretraga

Generisanje kombinatornih objekata

Problemi se često mogu rešiti iscrpnom pretragom (grubom silom), 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. U ovom poglavlju ćemo proučiti mehanizme njihovog sistematičnog generisanja. Naglasimo da po pravilu ovakvih objekata ima eksponencijalno mnogo u odnosu na veličinu ulaza, tako da su svi algoritmi praktično neupotrebljivi osim za veoma male dimenzije ulaza.

Objekti se obično predstavljaju n-torkama brojeva, pri čemu se isti objekti mogu torkama modelovati na različite načine. Na primer, svaki podskup skupa $\{a_1, \ldots, a_n\}$ 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). Na primer, torka $(2, 3, 5)$ jednoznačno određuje podskup $\{a_2, a_3, a_5\}$. Drugi način da se podskupovi predstave su n-torke logičkih vrednosti ili vrednost 0-1. Ako je $n = 5$, torka $(0, 1, 1, 0, 1)$ označava skup $\{a_2, a_3, a_5\}$.

Svi objekti se obično mogu predstaviti drvetom i to drvo odgovara procesu njihovog generisanja tj. obilaska (ono se ne pravi eksplicitno, u memoriji, ali nam pomaže da razumemo i organizujemo postupak pretrage). Obilazak drveta se najjednostavnije izvodi u dubinu (često rekurzivno). Za prvu navedenu reprezentaciju podskupova drvo je dato na slici 8. Svaki čvor drveta odgovara jednom podskupu, pri čemu se odgovarajuća torka očitava na granama puta koji vodi od korena do tog čvora.



Slika 8: Svi podskupovi četvoročlanog skupa - svaki čvor drveta odgovara jednom podskupu

Za drugu navedenu reprezentaciju podskupova drvo je dato na slici 9. Samo listovi drveta odgovaraju podskupovima, pri čemu se odgovarajuća torka očitava na granama puta koji vodi od korena do tog čvora.

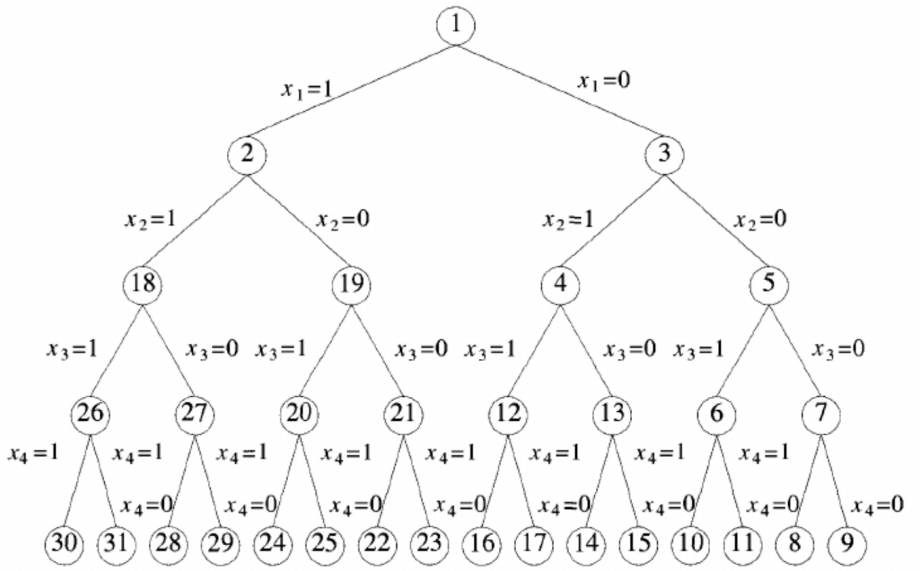
Primetimo da oba drveta sadrže 2^n čvorova kojima se predstavljaju podskupovi.

Prilikom generisanja objekata često je poželjno rešati ih određenim redom. S obzirom na to to da se svi kombinatorni objekti predstavljaju određenim torkama (konačnim nizovima), prirodan poredak među njima je *leksikografski poredak* (koji se koristi za utvrđivanje redosleda reči u rečniku). Podsetimo se, torka $a_0 \dots a_{m-1}$ leksikografski prethodi torci $b_0 \dots b_{n-1}$ akko postoji neki indeks i takav da za svako $0 \leq j < i$ važi $a_j = b_j$ i važi ili da je $a_i < b_i$ ili da je $i = m < n$. Na primer važi da je $11 < 112 < 221$ (ovde je $i = 2$, a zatim $i = 0$).

Na primer, ako podskupove skupa $\{1, 2, 3\}$ predstavimo na prvi način, torkama u kojima su elementi uređeni rastuće, leksikografski poredak bi bio $\emptyset$, $\{1\}$, $\{1, 2\}$, $\{1, 2, 3\}$, $\{1, 3\}$, $\{2\}$, $\{2, 3\}$, $\{3\}$. Ako bismo ih predstavljali na drugi način, torkama u kojima se nulama i jedinicama određuje da li je neki element uključen u podskup, leksikografski redosled bi bio: 000 ($\emptyset$), 001 ($\{3\}$), 010($\{2\}$), 011($\{2, 3\}$), 100($\{1\}$), 101($\{1, 3\}$), 110($\{1, 2\}$) i 111($\{1, 2, 3\}$).

Sve kombinacije dužine 3

Kada je dužina kombinatornih objekata mala, onda se oni mogu generisati i ugnežđenim petljama. Na primer, razmotrimo generisanje svih tročlanih kombinacija elemenata skupa $\{1, 2, \dots, n\}$.



Slika 9: Svi podskupovi četvoročlanog skupa - svaki list drveta odgovara jednom podskupu

Problem: Napiši funkciju kojom se nabrajaju sve tročlane kombinacije bez ponavljanja elemenata skupa $\{1, 2, \ldots, n\}$. Na primer, za $n = 4$, kombinacije su 123, 124, 134 i 234 (pretpostavlja se da se svaka kombinacija predstavlja tročlanim, sortiranim nizom brojeva). Pretpostaviti da je na raspolaganju funkcija `void obradi(int kombinacija[3]);` koju treba pozvati da bi se obradila svaka od kombinacija.

```
void troclaneKombinacije(int n) {
   int kombinacija[3];
   for (int i = 1; i <= n-2; i++) {
      kombinacija[0] = i;
      for (int j = i+1; j <= n-1; j++) {
         kombinacija[1] = j;
         for (int k = j+1; k <= n; k++) {
            kombinacija[2] = k;
            obradi(kombinacija);
         }
      }
   }
}
```

Veoma slično bi se mogle generisati i obraditi i sve tročlane varijacije.

Svi podskupovi

Razmotrimo sada problem nabrajanja svih podskupova datog skupa. Ako se elementi skupa pamte u nizu, tada se potpuno isti algoritam primenjuje i za nabrajanje svih multi-podskupova (ponavljanje elemenata nije bitno).

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).

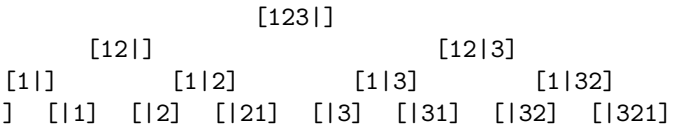
Iako jezik C++, kao i mnogi drugi savremeni jezici, pruža tip za reprezentovanje skupova, implementacija je jednostavnija i efikasnija ako se elementi skupa čuvaju u nizu ili vektoru.

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.

```
// procedura određuje i obrađuje sve moguće skupove koji se dobijaju
// tako što se na elemente prosleđenog podskupa dodaju elementi
// podskupova prosleđenog skupa
void obradiSvePodskupove(const vector<int>& skup,
                         const vector<int>& podskup) {
  // skup je prazan
  if (skup.size() == 0)
    // na prosleđeni podskup možemo dodati samo prazan skup
    ispisi(podskup);
  else {
    // izdvajamo i uklanjamo proizvoljan element skupa
    int x = skup.back();
    vector<int> smanjenSkup = skup;
    smanjenSkup.pop_back();
    // rekurzivnim pozivom u podskup obrađujemo sve podskupove
    // skupa koji ne uključuju izdvojeni element
    vector<int> podskupBez = podskup;
    obradiSvePodskupove(smanjenSkup, podskupBez);
    // rekurzivnim pozivom u podskup obrađujemo sve podskupove
```

```
    // skupa koji uključuju izdvojeni element
    vector<int> podskupSa = podskup;
    podskupSa.push_back(x);
    obradiSvePodskupove(smanjenSkup, podskupSa);
  }
}

void obradiSvePodskupove(const vector<int>& skup) {
  // krećemo od praznog podskupa
  vector<int> podskup;
  // prazan skup proširujemo svim podskupovima datog skupa
  obradiSvePodskupove(skup, podskup);
}
```

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.



U prethodnoj implementaciji smo se direktno oslonili na mogućnosti dodavanja i izbacivanja elemenata iz vektora i tako gradili manje i veće skupove. Ta implementacija je veoma loša, jer smo prilikom svakog rekurzivnog poziva pravili kopije skupova, što je veoma neefikasna operacija. 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.

```
// procedura određuje i obrađuje sve moguće skupove koji se dobijaju
// tako što se na elemente prosleđenog podskupa dodaju elementi
// podskupova prosleđenog skupa
void obradiSvePodskupove(vector<int>& skup, vector<int>& podskup) {
   // skup je prazan
   if (skup.size() == 0)
      // na prosleđeni podskup možemo dodati samo prazan skup
      obradi(podskup);
   else {
      // izdvajamo i uklanjamo proizvoljan element skupa
      int x = skup.back();
      skup.pop_back();
      // u podskup dodajemo sve podskupove skupa bez izdvojenog
      // elementa
      obradiSvePodskupove(skup, podskup);
```

```
      // u podskup uključujemo izdvojeni element i zatim sve
      // podskupove skupa bez izdvojenog elementa
      podskup.push_back(x);
      obradiSvePodskupove(skup, podskup);
      // vraćamo skup i podskup u početno stanje
      podskup.pop_back();
      skup.push_back(x);
   }
}

void obradiSvePodskupove(vector<int>& skup) {
   // krećemo od praznog podskupa
   vector<int> podskup;
   // efikasnosti radi rezervišemo potrebnu memoriju za najveći podskup
   podskup.reserve(skup.size());
   // prazan skup proširujemo svim podskupovima datog skupa
   obradiSvePodskupove(skup, podskup);
}
```

Još bolje rešenje možemo dobiti tako što se izbacivanje elemenata iz skupa ne vrši tako što se efektivno gradi manji skup (kraći vektor) koji će se prosleđivati kroz rekurziju, nego tako što 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. Pošto nam je sada svejedno da li će izbačeni elementi biti na kraju ili na početku, da bismo nabrojali podskupove u rastućem leksikografskom poretku, biraćemo da brojač i razdvaja niz *skup* na dva dela: elementi na pozicijama $[0, i)$ su izbačeni a na pozicijama $[i, n)$ su preostali (gde je n broj elemenata skupa). I dužinu vektora u kome se čuva podskup ćemo u startu postaviti na n elemenata, pri čemu ćemo smatrati 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. Tako dobijamo narednu implementaciju.

```
// popunjava niz podskup od pozicije j naviše svim podskupovima
// skupa elemenata vektora skup od pozicije i pa naviše
void obradiSvePodskupove(const vector<int>& skup, int i,
                         vector<int>& podskup, int j) {
  // u skupu nema više preostalih elemenata
  if (i == skup.size()) {
    // jedini podskup praznog skupa je prazan i
    // njegovim dodavanjem podskup se ne menja
    obradi(podskup, j);
  } else {
    // izbacujemo element i iz skupa, a podskup ne menjamo
    obradiSvePodskupove(skup, i + 1, podskup, j);
    // izbacujemo element i iz skupa i prebacujemo ga u podskup
    podskup[j] = skup[i];
    obradiSvePodskupove(skup, i + 1, podskup, j + 1);
```

```
  }
}

void obradiSvePodskupove(const vector<int>& skup) {
  // alociramo potreban prostor za najveći podskup
  vector<int> podskup(skup.size());
  // prazan podskup dopunjavamo elementima podskupova celog skupa
  obradiSvePodskupove(skup, 0, podskup, 0);
}
```

Funkcija u ovom stilu se može isprogramirati i pomoću klasičnih statički ili dinamički alociranih nizova i nije ni po čemu specifična za C++. Kod generisanja kombinatornih objekata u nastavku ćemo u startu podrazumevati ovakav pristup.

Još jedan način da se nabroje svi kombinatorni objekti je da se definiše funkcija koja na osnovu datog objekta generiše sledeći (ili eventualno prethodni) u nekom redosledu (najčešće leksikografskom). Definišimo tako funkciju koja na osnovu datog, određuje naredni podskup u leksikografskom redosledu.

Napišimo, na primer, leksikografski uređenu listu svih podskupova skupa brojeva od 1 do 4.

```
-
1
12
123
1234
124
13
134
14
2
23
234
24
3
34
4
```

Možemo primetiti da postoje dva načina da se dođe do narednog podskupa. Analizirajmo ove skupove u istom redosledu, grupisane i na osnovu broja elemenata.

```
- 1   12   123   1234
          124
      13   134
      14
  2   23   234
```

```
   24
3  34
4
```

Jedan način je *proširivanje* kada se naredni podskup dobija dodavanjem nekog elementa u prethodni. To su koraci u prethodnoj tabeli kod kojih se prelazi iz jedne u narednu kolonu. 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 (izuzetak je prazan skup, koji se proširuje elementom 1). Jedini slučaj kada proširivanje nije moguće je kada je poslednji element podskupa najveći mogući (u našem primeru to je 4).

Drugi način je *skraćivanje* kada se naredni element dobija uklanjanjem nekih elemenata iz podskupa i izmenom preostalih elemenata. To su koraci u prethodnoj tabeli kod kojih se prelazi sa kraja jedne u narednu vrstu. U ovom slučaju skraćivanje funkcioniše tako što se iz podskupa izbaci završni najveći element, a zatim se najveći od preostalih elemenata uveća za 1 (on ne može biti najveći, jer su elementi unutar svake kombinacije strogo rastući). Ako nakon izbacivanja najvećeg elementa ostane prazan skup, naredna kombinacija ne postoji.

```
// na osnovu datog podskupa skupa {1, ..., n} određuje
// leksikografski naredni podskup i vraća da li takav
// podskup postoji
bool sledeciPodskup(vector<int>& podskup, int n) {
   // specijalni slučaj proširivanja praznog skupa
   if (podskup.empty()) {
      podskup.push_back(1);
      // podskup je uspešno pronađen
      return true;
   }
   // proširivanje
   if (podskup.back() < n) {
      // u podskup dodajemo element koji je za 1 veći od
      // trenutno najvećeg elementa
      podskup.push_back(podskup.back() + 1);
      // podskup je uspešno pronađen
      return true;
   }

   // skraćivanje
   // uklanjamo poslednji najveći element
   podskup.pop_back();
   // ako nema preostalih elemenata ne postoji naredni podskup
   if (podskup.empty())
      return false;
   // najveći od preostalih elemenata uvećavamo za 1
   podskup.back()++;
   // podskup je uspešno pronađen
```

```
   return true;
}

// obrada svih podskupova skupa {1, ..., n}
void obradiPodskupove(int n) {
  // tekući podskup
  vector<int> podskup;
  // obrađujemo podskupove redom, sve dok je moguće pronaći
  // leksikografski sledeći podskup
  do {
    obradi(podskup);
  } while (sledeciPodskup(podskup, n));
}
```

Naravno, umesto u vektoru, podskup smo mogli čuvati i u običnom nizu, ali je onda potrebno da prosleđujemo i dimenziju podskupa.

```
// na osnovu datog podskupa skupa {1, ..., n} određuje
// leksikografski naredni podskup i vraća da li takav
// podskup postoji.
// Tekući podskup je smešten u nizu dužine i
bool sledeciPodskup(int podskup[], int& i, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (i == 0) {
    podskup[i++] = 1;
    return true;
  }

  // proširivanje
  if (podskup[i-1] < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup[i] = podskup[i-1] + 1;
    i++;
    return true;
  }

  // skraćivanje

  // izbacujemo najveći element iz podskupa
  i--;
  // ako nema preostalih elemenata, naredni podskup ne postoji
  if (i == 0)
    return false;

  // najveći od preostalih elemenata uvećavamo za 1
  podskup[i-1]++;
  return true;
}
```

Sve varijacije

Problem: Definisati funkciju koja obrađuje sve varijacije sa ponavljanjem dužine k skupa $\{1, \ldots, n\}$. Na primer, sve varijacije dužine 3 elemenata $\{0, 1\}$ su:

```
000
001
010
011
100
101
110
101
```

Recimo i da se problem obrade svih podskupova skupa $\{1, \ldots, n\}$ može svesti na problem određivanja svih varijacija dužine n skupa $\{0, 1\}$ - svaka varijacija jednoznačno odgovara jednom podskupu (određenom jedinicama u varijaciji).

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).

```
// sve varijacije dužine k elemenata skupa {1, ..., n}
// Dati niz varijacija dužine varijacije.size() - k
// se dopunjuje svim mogućim varijacijama sa ponavljanjem
// dužine k skupa {1, ..., n} i sve tako
// dobijene varijacije se obrađuju
void obradiSveVarijacije(int k, int n,
                         vector<int>& varijacija) {
  // k je 0, pa je jedina varijacija dužine nula prazna i njenim
  // dodavanjem na polazni niz on se ne menja
  if (k == 0)
    obradi(varijacija);
  else
    // na tekuću poziciju postavljamo sve moguće vrednosti od 1 do n i
    // dobijeni niz onda rekurzivno proširujemo
    for (int nn = 1; nn <= n; nn++) {
       varijacija[varijacija.size() - k] = nn;
       obradiSveVarijacije(k-1, n, varijacija);
    }
}

void obradiSveVarijacije(int k, int n) {
  vector<int> varijacija(k);
  obradiSveVarijacije(k, n, varijacija);
```

```
}
```

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* (engl. turning point). Na primer, ako nabrajamo varijacije skupa $\{1, 2, 3\}$ dužine 5 naredna varijacija za varijaciju 21332 je 21333 (prelomna tačka je pozicija 4, koja je poslednja pozicija u nizu), dok je njoj naredna varijacija 22111 (prelomna tačka je pozicija 1 na kojoj se nalazi element 1). Niz 33333 nema prelomnu tačku, pa samim tim ni leksikografski sledeću varijaciju.

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) {
   // od kraja varijacije tražimo prvi element koji se može povećati
   int i;
   int k = varijacija.size();
   for (i = k-1; i >= 0 && varijacija[i] == n; i--)
      varijacija[i] = 1;
   // svi elementi su jednaki n - ne postoji naredna varijacija
   if (i < 0) return false;
   // uvećavamo element koji je moguće uvećati
   varijacija[i]++;
   return true;
}

void obradiSveVarijacije(int k, int n) {
   // krećemo od varijacije 11...11 - ona je leksikografski najmanja
   vector<int> varijacija(k, 1);
   // obrađujemo redom varijacije dok god postoji leksikografski
   // sledeća
   do {
     obradi(varijacija);
   } while(sledecaVarijacija(n, varijacija));
}
```

Svi binarni zapisi bez uzastopnih jedinica

Vežbe radi razmotrimo i narednu varijaciju prethodnog problema.

Problem: Definisati funkciju koja obrađuje sve binarne zapisse dužine n kom se ne javljaju dve uzastopne jedinice.

Zadatak se može rešiti prilagođavanjem prethodne funkcije za generisanje svih varijacija. Binarne zapise možemo predstaviti vektorima, ali i niskama (isto smo mogli uraditi i u prethodnim zadacima). Nisku ćemo u startu alocirati na dužinu n, a onda malo po malo popunjavati tokom rekurzije. Dužinu popunjenog dela niske tj. tekuću poziciju na koju treba upisati naredni element možemo prosleđivati kroz funkciju (isto smo mogli uraditi i kod generisanja klasičnih varijacija). Kada i dostigne dužinu niske, cela niska je popunjena, pa dobijenu varijaciju obrađujemo. U suprotnom, na poziciju i uvek možemo dopisati nulu i rekurzivno nastaviti sa produžavanjem tako dobijene niske. Sa druge strane, jedinicu možemo upisati samo ako prethodni karakter nije jedinica (u suprotnom bismo dobili dve uzastopne jedinice). To se dešava ili kada nema prethodne cifre (kada je $i = 0$) ili kada je prethodna cifra (na poziciji $i - 1$) različita od jedinice.

```
void obradiSveBinarneBez11(string& binarni, int i) {
  if (i == binarni.size())
    obradi(binarni);
  else {
    binarni[i] = '0';
    ispisiSveBinarneBez11(binarni, i+1);
    if (i == 0 || binarni[i-1] != '1') {
      binarni[i] = '1';
      obradiSveBinarneBez11(binarni, i+1);
    }
  }
}

void ispisiSveBinarneBez11(int n) {
  string binarni(n, '0');
  obradiSveBinarneBez11(binarni, 0);
}
```

Algoritam kojim se određuje sledeću nisku bez susednih jedinica u leksikografskom redosledu predstavljaće modifikaciju algoritma kojim se određuje sledeća varijacija u leksikografskom redosledu. Podsetimo se, krećemo od kraja niza, pronalazimo prvu poziciju na kojoj se nalazi element čija vrednost nije trenutno maksimalna (ako ona postoji), uvećavamo je, i nakon toga sve elemente iza nje postavljamo na minimalnu vrednost (u našem kontekstu maksimalna vrednost je 1, a minimalna 0). Ovo možemo ostvariti i u jednom prolasku tako što krećući se unazad sve maksimalne vrednosti odmah postavljamo na nulu.

Modifikujmo sada ovaj algoritam tako da radi za nizove koji nemaju dve uzastopne jedinice. Ako se nakon primene prethodnog algoritma dogodilo da je uvećana (postavljena na jedinicu) cifra ispred koje ne stoji jedinica, to je rešenje koje smo tražili. Na primer, sledeći niz u odnosu na niz 01001 je 01010. Međutim, ako zatražimo naredni element, dobićemo 01011, a to je element koji nije

dopušten. Nastavljanjem dalje dobijamo 01100, što je takođe element koji nije dopušten, nakon togaređaju se elementi od 01101, do 01111, koji su svi nedopušteni da bismo na kraju dobili 10000, što je zapravo naredni element u odnosu na 01010. Dakle, opet se krećemo sa kraja niza i upisujemo nule sve dok se na trenutnoj ili na prethodnoj poziciji u nizu nalazi jedinica. Na kraju, na poziciji na kojoj smo se zaustavili i nismo upisali nulu (ako takva postoji) upisujemo jedinicu (to je pozicija na kojoj piše nula i ispred nje ili nema ništa ili piše nula). Ako takva pozicija ne postoji, onda je trenutni niz leksikografski najveći.

```
bool sledecaVarijacijaBezUzastopnihJedinica(string& s){
  int n = s.length();
  int i = n - 1;
  while ((i >= 0 && s[i] == '1') ||
         (i > 0 && s[i-1] == '1'))
    s[i--] = '0';
  if (i < 0)
    return false;
  s[i] = '1';
  return true;
}
```

Sve permutacije

Problem: Definisati proceduru koja nabraja i obrađuje sve permutacije skupa $\{1, 2, \ldots, n\}$. Na primer, za $n = 3$, permutacije su 123, 132, 213, 231, 312 i 321.

Rekurzivno generisanje permutacija u leksikografskom redosledu je veoma komplikovano, tako da ćemo se odreći uslova da permutacije moraju biti poređane leksikografski.

U tom slučaju možemo postupiti na sledeći način. 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 (pošto je ona već na mestu 0 nema potrebe ništa dodatno raditi) 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. Na primer, ako je niz na početku 123, onda menjamo element 3 sa elementom 1, dobijamo 321 i pozivamo rekurzivno generisanje permutacija niza 32 sa fiksiranim elementom 1 na kraju. Zatim u početnom nizu menjamo element 3 sa elementom 2, dobijamo 132 i pozivamo rekurzivno generisanje permutacija niza 13 sa

fiksiranim elementom 2 na kraju. Zatim u početnom nizu menjamo element 3 sa samim sobom, dobijamo 123 i pozivamo rekurzivno generisanje permutacija niza 12 sa fiksiranim elementom 3 na kraju. 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. Naime, 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. Na osnovu toga znamo i da će invarijanta funkcije biti održana i nije potrebno uraditi ništa dodatno nakon petlje.

```
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);
}
```

Implementirajmo sada funkciju koja pronalazi narednu permutaciju u leksikografskom poretku.

Razmotrimo permutaciju 13542. Zamenom elementa 2 i 4 bi se dobila permutacija 13524 koja je leksikografski manja od polazne i to nam ne odgovara. Slično bi se desilo i da se zamene elementi 5 i 4. Činjenica da je niz 542 strogo opadajući nam govori da nije moguć ni na koji način razmeniti ta tri elementa da se dobije leksikografski veća permutacija, tj. da je ovo najveća permutacija koja počinje sa 13. Dakle, naredna permutacija će biti leksikografski najmanja

permutacija koja počinje sa 14, a to je 14235.

Dakle, 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. U našem primeru $a_i = 3$. 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 $a_i < a_j$ (opet linearnom pretragom) i razmenjujemo elemente na pozicijama i i j. U našem primeru $a_j = 4$ i nakon razmene dobijamo permutaciju 14532. Pošto je ovom razmenom rep iza pozicije i i dalje striktno opadajući, da bismo dobili željenu leksikografski najmanju permutaciju koja počinje sa 14, potrebno je obrnuti redosled njegovih elemenata.

```
bool sledecaPermutacija(vector<int>& permutacija){
  int n = permutacija.size();

  // linearnom pretragom pronalazimo prvu poziciju i takvu da
  // je permutacija[i] > permutacija[i+1]
  int i = n - 2;
  while (i >= 0 && permutacija[i] > permutacija[i+1])
    i--;
  // ako takve pozicije nema, permutacija a je leksikografski maksimalna
  if (i < 0) return false;
  // linearnom pretragom pronalazimo prvu poziciju j takvu da
  // je permutacija[j] > permutacija[i]
  int j = n - 1;
  while (permutacija[j] < permutacija[i])
    j--;
  // razmenjujemo elemente na pozicijama i i j
  swap(permutacija[i], permutacija[j]);
  // obrćemo deo niza od pozicije i+1 do kraja
  for (j = n - 1, i++; i < j; i++, j--)
    swap(permutacija[i], permutacija[j]);
  return true;
}
```

Sve kombinacije

Problem: Definisati proceduru koja nabraja i obrađuje sve kombinacije bez ponavljanja dužine k skupa $\{1, 2, \ldots, n\}$, za $k \leq n$. Na primer, za $k = 3$ i $n = 5$ kombinacije su 123, 124, 125, 134, 135, 145, 234, 235, 245, 345.

Da bi se izbeglo višestruko generisanje kombinacija koje su suštinski identične nametnućemo uslov da je svaka kombinacija predstavljena rastući sortiranim nizom brojeva. Kao i u prethodnim slučajevima 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. Zapravo, ovo gornje ograničenje može još da se smanji. 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.

```
// niz kombinacije dužine k na pozicijama [0, i) sadrži uređen
// niz elemenata. Procedura na sve moguće načine dopunjava taj niz
// elementima iz skupa [1, n] tako da niz bude uređen rastući
void obradiSveKombinacije(vector<int>& kombinacija, int i, int n) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }
  // određujemo raspon elemenata na poziciji i
  int pocetak = i == 0 ? 1 : kombinacija[i-1]+1;
  int kraj = n + i - k + 1;
  // jedan po jedan element upisujemo na poziciju i, pa
  // nastavljamo generisanje rekurzivno
  for (int x = pocetak; x <= kraj; x++) {
    kombinacija[i] = x;
    obradiSveKombinacije(kombinacija, i+1, n);
  }
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacije(kombinacija, 0, n);
}
```

Postoji način da izbegnemo rekurzivne pozive u petlji. Tokom rekurzije možemo da čuvamo informaciju o tome koji je raspon elemenata kojim se proširuje niz. Znamo da su to elementi skupa $\{1,\dots,n\}$, međutim, pošto su kombinacije sortirane rastuće skup kandidata je uži. U prethodnom programu smo najmanju vrednost za poziciju i određivali na osnovu vrednosti sa pozicije $i-1$, međutim, alternativno možemo i eksplicitno da održavamo promenljive n_{min} i n_{max} koje određuju skup $\{n_{min},\dots,n_{max}\}$ čiji se elementi raspoređuju u kombinaciji na pozicijama iz intervala $[i,k)$. Ako je taj interval prazan, kombinacija je popunjena i može se obraditi. U suprotnom, ako je $n_{min} > n_{max}$, tada ne postoji vrednost koju je moguće staviti na poziciju i, pa možemo izaći iz rekurzije, jer

se trenutna kombinacija ne može popuniti do kraja. U suprotnom možemo razmotriti dve mogućnosti. Prvo na poziciju i možemo postaviti element n_{min} i rekurzivno izvršiti popunjavanje niza od pozicije $i+1$, a drugo možemo taj element preskočiti i u rekurzivnom pozivu ponovo zahtevati da se popuni pozicija i. U oba slučaja se skup elemenata sužava na $\{n_{min}+1, \ldots, n_{max}\}$.

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

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }

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

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

Analogno prethodnoj implementaciji, pretragu možemo saseći i malo ranije. Naime, pošto su ponavljanja zabranjena kada je broj elemenata tog skupa (a to je $n_{max} - n_{min} + 1$) manji od broja preostalih pozicija koje treba popuniti (a to je $k - i$), već tada možemo saseći pretragu, jer ne postoji mogućnost da se kombinacija uspešno dopuni do kraja. Izlaz iz rekurzije u slučaju nemogućnosti proširivanja tekuće kombinacije se tada može zameniti sa

```
  if (n_max - n_min + 1 < k - i)
    return;
```

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, pretposlednji ako nije jednak $n-1$ 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 (ako su sve vrednosti na svojim

maksimumima), 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. Na primer, ako je $k = 6$, tada je naredna kombinacija kombinaciji 1256, kombinacija 1345 - prelomna vrednost je 2 i ona se može uvećati na 3, nakon čega slažemo redom elemente za po jedan veće.

```
bool sledecaKombinacija(int n, vector<int>& kombinacija) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // krećemo od kraja i tražimo prvu poziciju koja nije na maksimumu
  // tj. koja se može povećati. Maksimumi od kraja su n, n-1, n-2, ...
  int i;
  for (i = k-1; i >= 0 && kombinacija[i] == n; i--, n--)
    ;
  // ako takva pozicija ne postoji, tekuća kombinacija je maksimalna
  if (i < 0)
    return false;
  // uvećavamo poslednji element koji se može povećati
  kombinacija[i]++;
  // iza njega slažemo redom brojeve za jedan veće
  for (i++; i < k; i++)
    kombinacija[i] = kombinacija[i-1] + 1;
  return true;
}

void obradiSveKombinacije(int k, int n) {
  // krecemo od kombinacije 1, 2, ..., k
  vector<int> kombinacija(k);
  for (int i = 0; i < k; i++)
    kombinacija[i] = i + 1;

  // obradjujemo kombinacije dokle god postoji sledeca
  do {
    obradi(kombinacija);
  } while (sledecaKombinacija(n, kombinacija));
}
```

Postoji i mala optimizacija prethodnog postupka. Naime, ako znamo vrednost prelomne tačke u jednom koraku, bez ponovne pretrage možemo odrediti vrednost prelomne tačke u narednom koraku. Ključno pitanje je to da li nakon uvećanja prelomne vrednosti ona dostiže svoj maksimum.

- Ako dostiže tj. ako posle uvećanja važi $komb_i = n - k + i + 1$, tada se posle uvećanja prelomne vrednosti, redom iza prelomne tačke moraju

naći elementi koji su svi na svojim maksimumima, međutim, to je već slučaj tako da nije potrebno ponovo ih ažurirati. Naredna prelomna tačka je neposredno ispred tekuće prelomne tačke. Na primer, naredna kombinacija za 1356 je 1456. Prelomna tačka je $i = 1$, važi da je $komb_1 = 3 = 6 - 4 + 1$ i dovoljno je samo uvećati element 3 na 4. Pošto su elementi od 4, 5 i 6, na svojoj maksimalnoj vrednosti, znamo da je naredna prelomna vrednost 1, pa je naredna kombinacija 2345.

- Ako nakon uvećanja prelomna vrednost ne dostiže svoj maksimum tj. ako nakon uvećanja važi $komb_i < n - k + i + 1$, onda je nakon uvećanja i popunjavanja niza do kraja poslednji element sigurno ispod svoje maksimalne vrednosti, tako da je naredna prelomna tačka poslednja pozicija u nizu.

```
void obradiSveKombinacije(int k, int n) {
   // krecemo od kombinacije 1, 2, ..., k
   vector<int> kombinacija(k);
   for (int i = 0; i < k; i++)
     kombinacija[i] = i + 1;

   // obrađujemo prvu kombinaciju
   obradi(kombinacija);

   // specijalno obrađujemo slučaj n = k kada
   // nema drugih kombinacija
   if (n == k) return;

   // prva prelomna tačka je poslednja pozicija u nizu
   int i = k-1;
   while (i >= 0) {
      // ažuriramo kombinaciju
      kombinacija[i]++;
      // ako je uvećani prelomni element dostigao maksimum
      if (kombinacija[i] == n - k + i + 1)
        // naredna prelomna tačka je neposredno pre njega
        i--;
      else {
        // popunjavamo niz do kraja
        for (int j = i+1; j < k; j++)
           kombinacija[j] = kombinacija[j-1] + 1;
        // naredna prelomna tačka je poslednji element niza
        i = k-1;
      }
      // obrađujemo dobijenu kombinaciju
      obradi(kombinacija);
   }
}
```

Interesantno, ova "optimizacija" ne donosi nikakvu značajnu dobit i nema praktičnih implikacija. Ako obrada uključuje bilo kakvu netrivijalnu operaciju ili

ispis kombinacije na ekran, obrada potpuno dominira vremenom generisanja. Ako je obrada trivijalna (na primer, samo uvećanje globalnog brojača za jedan), tada se generisanje svih kombinacija za $n = 10$, $k = 50$ izvršava za oko 20 sekundi, a optimizovanoj verziji je potrebno oko 17 sekundi. Razlog tome je to što je u većini slučajeva prelomna tačka poslednji element ili je vrlo blizu desnog kraja, pa je linearna pretraga brza. Rekurzivna verzija za isti zadatak zahteva oko 34 sekunde.

Sve kombinacije sa ponavljanjem

Problem: Definisati proceduru koja nabraja i obrađuje sve kombinacije sa ponavljanjem dužine k skupa $\{1, 2, \ldots, n\}$, za $k \leq n$. Na primer, za $k = 3$ i $n = 3$ kombinacije sa ponavljenjem su 111, 112, 113, 122, 123, 133, 222, 223, 233, 333.

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. Razmotrimo, na primer, rekurzivno rešenje. 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) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }

  // ako ne postoji element koji možemo upisati na poziciju i,
  // kombinacija se ne može produžiti
  if (n_min > n_max)
    return;

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

```
void obradiSveKombinacijeSaPonavljanjem(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacijeSaPonavljanjem(kombinacija, 0, 1, n);
}
```

Sve particije

Problem: Particija pozitivnog prirodnog broja n je predstavljanje broja n kao zbira nekoliko pozitivnih prirodnih brojeva pri čemu je redosled sabiraka nebitan ($1+2$ i $2+1$ predstavljaju jednu te istu particiju broja 3). Na primer particije broja 4 su $1+1+1+1$, $2+1+1$, $2+2$, $3+1$, 4. Napisati program koji ispisuje sve particije datog broja n sortirane leksikografski (bilo rastuće, bilo opadajuće).

Svaka particija ima svoj prvi sabirak. Svakoj particiji broja n kojoj je prvi sabirak s (pri čemu je $1 \leq s \leq n$) jednoznačno odgovara neka particija broja $n-s$, što ukazuje da se problem može rešavati induktivno-rekurzivnom konstrukcijom. Pošto je sabiranje komutativno, da ne bismo suštinski iste particije brojali više puta nametnućemo uslov da sabirci u svakoj particiji budu sortirani (na primer, nerastuće). Zato, ako je prvi sabirak s, svi sabirci iza njega moraju da budu manji ili jednaki od s. Zato nam nije dovoljno samo da umemo da 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} (to je poslednja, najmanja od svih vrednosti na pozicijama $[0, i)$, ali je označena kao max jer predstavlja gornju granicu vrednosti za element na poziciji i). 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}, a prirodno je da moraju da budu i manji ili jednaki od n (jer prvi sabirak ne može biti veći od zbira prirodnih brojeva). 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);
```

```
    }
  }
}
```

U glavnoj funkciji ćemo alocirati niz dužine n (jer najduža particija ima n sabiraka koji su svi jednaki 1) i zahtevaćemo da se taj niz popuni počevši od pozicije 0 particijama broja n u kojima su svi sabirci manji ili jednaki n.

```
void obradiParticije(int n) {
  vector<int> particija(n);
  obradiParticije(n, n, particija, 0);
}
```

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 $s_{max}-1$. U zavisnosti od redosleda ova dva rekurzivna poziva određuje se da li će particije biti sortirane leksikografski rastuće ili opadajuće.

```
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);
    }
  }
}
```

Svi izrazi

Problem: Napisati program koji ispisuje sve izraze koji se mogu dobiti od datog niza celobrojnih vrednosti primenom četiri osnovna aritmetička operatora. Svaka vrednost iz liste se u izrazu može javiti najviše jednom. Na primer, ako za vrednosti 1 i 2, dobijaju se izrazi 2, 1, $(2+1)$, $(2-1)$, $(2*1)$, $(2/1)$, $(1+2)$, $(1-2)$, $(1*2)$, $(1/2)$.

Zadatak ćemo rešavati kroz dve faze. U prvoj ćemo određivati sve celobrojne vrednosti iz datog niza koje će učestvovati u datom izrazu, a u drugoj ćemo kreirati sve izraze u kojima se javljaju te odabrane vrednosti baš u tom redosledu.

Biranje svih vrednosti odgovara biranju svih mogućih podskupova datog skupa, međutim, pošto je i redosled bitan, za svaki odabran skup određivaćemo i sve permutacije. Proceduru za generisanje svih podskupova i proceduru za generisanje svih permutacija smo već opisali. U ranijim implementacijama se prilikom generisanja svakog objekta pozivala određena globalna funkcije (nazivali smo je `obradi`), a ovaj put ćemo napraviti malo fleksibilniju implementaciju i procedure koje se pozivaju nad svim generisanim objektima ćemo prosleđivati kao parametre. Mogli bismo upotrebiti pokazivače na funkcije (koji se koriste na isti način kao u C-u), ali malo fleksibilnije rešenje dobijamo ako koristimo tip `function<T>`, jer tada u pozivu možemo koristiti i anonimne (lambda) funkcije.

```
// tip procedure koja obrađuje deo vektora
// (ona je tipa void, a prima vector<int> i int)
typedef function<void (vector<int>&, int)> FunPtr;

// funkcija permutuje deo vektora vals dužine n na pozicijama [0,k),
// pozivajući funkciju f za svaku permutaciju
void permutations(vector<int>& vals, int n, int k, const FunPtr& fun) {
  if (k == 0)
    fun(vals, n);
  else {
    for (int i = 0; i < k; i++) {
      swap(vals[i], vals[k-1]);
      permutations(vals, n, k-1, fun);
      swap(vals[k-1], vals[i]);
    }
  }
}

// funkcija permutuje vektor vals dužine n, pozivajući funkciju
// fun za svaku permutaciju
void permutations(vector<int>& vals, int n, const FunPtr& fun) {
  permutations(vals, n, n, fun);
}

// funkcija dopunjava vektor chosenVals od pozicije j svim podskupovima
// elemenata vektora vals na pozicijama [k, n), pozivajući funkciju fun
// za svaki generisani podskup
void subsets(vector<int>& vals, int k, vector<int>& chosenVals, int j,
             const FunPtr& fun) {
  if (k == vals.size()) {
    fun(chosenVals, j);
  } else {
    subsets(vals, k+1, chosenVals, j, fun);
    chosenVals[j] = vals[k];
    subsets(vals, k+1, chosenVals, j+1, fun);
  }
}
```

```
// Funkcija generiše sve podskupove vektora vals i poziva funkciju fun
// za svaki generisani podskup
void subsets(vector<int>& vals, const FunPtr& fun) {
  vector<int> chosenVals(vals.size());
  subsets(vals, 0, chosenVals, 0, fun);
}
```

Sve permutacije svih podskupova bismo mogli ispisati na sledeći način.

```
// ispisuje elemente vektora a duzine n
void print(const vector<int>& a, int n) {
  for (int i = 0; i < n; i++)
    cout << a[i] << " ";
  cout << endl;
}

int main() {
  vector<int> a{1, 3, 7, 10, 25, 50};
  // odredjujemo sve podskupove,
  // za svaki podskup određujemo sve permutacije,
  // a svaku permutaciju ispisujemo
  subsets(a, [](vector<int>& a, int n) {
                permutations(a, n, [](vector<int>& a, int n) {
                                       print(a, n);
                                   });
            });
  return 0;
}
```

Kada je fiksiran spisak i redosled brojeva koji će učestvovati u izrazima, možemo preći na generisanje samih izraza. Izraze ćemo predstavljati na uobičajeni način - binarnim drvetima. Razlikuju se dva tipa čvora: listovi u kojima se nalaze celobrojne vrednosti i unutrašnji čvorovi u kojima se nalaze operatori. Jedan veoma jednostavan (ali ne i najbolji) način da se predstave svi čvorovi je da se definiše jedan strukturni tip u kojem objedinjavamo podatke koji se koriste i u listovima i u unutrašnjim čvorovima. Da ne bismo brinuli o dealokaciji drveta, umesto običnih, koristićemo pametne pokazivače koje imamo na raspolaganju u jeziku C++.

Krenimo od reprezentacije operatora i pomoćnih funkcija za baratanje operatorima.

```
// podržana su četiri operatora: +, -, * i /
enum Op {Add, Sub, Mul, Div};

// štampanje operatora na standardni izlaz
void printOp(Op op) {
  switch(op) {
  case Add:
```

```
    cout << "+"; break;
  case Sub:
    cout << "-"; break;
  case Mul:
    cout << "*"; break;
  case Div:
    cout << "/"; break;
  }
}
```

Sada možemo definisati tipove izraza (listovi su obeleženi sa `Val`, dok su unutrašnji čvorovi obeleženi sa `App`).

```
// tipovi izraza
enum ExprType {Val, App};

// čvor izraza
struct Expr {
  ExprType type;
  int val;
  Op op;
  shared_ptr<Expr> left, right;
};

// pametni pokazivač na izraz
typedef shared_ptr<Expr> ExprPtr;
```

Definišemo i konstruktore, tj. funkcije koje kreiraju izraze na osnovu datih podataka.

```
// pomoćna funkcija za kreiranje čvora izraza
ExprPtr mkExpr() {
  return make_shared<Expr>();
}

// kreiranje lista sa datom celobrojnom vrednošću
ExprPtr mkVal(int x) {
  ExprPtr e = mkExpr();
  e->type = Val;
  e->val = x;
  return e;
}

// kreiranje unutrašnjeg čvora sa datim operatorom i datim potomcima
ExprPtr mkApp(Op op, ExprPtr op1, ExprPtr op2) {
  ExprPtr e = mkExpr();
  e->type = App;
  e->op = op;
  e->left = op1;
  e->right = op2;
```

```
  return e;
}
```

Jednostavno možemo definisati i funkciju za ispisivanje izraza u infiksnom obliku (izraz se štampa u potpuno zagrađenom obliku).

```
void printExpr(ExprPtr e) {
  if (e->type == Val)
    cout << e->val;
  else {
    cout << "(";
    printExpr(e->left);
    printOp(e->op);
    printExpr(e->right);
    cout << ")";
  }
}
```

Centralna funkcija u drugoj fazi ovog zadatka je generisanje svih izraza ako je dat vektor kojim je određen skup i redosled vrednosti u listovima. Operator u korenu izraza deli taj vektor na dve neprazne polovine: elemente koji se nalaze levo i elemente koji se nalaz desno od tog operatora. Pošto želimo da generišemo sve izraze, razmatramo sve mogućnosti. Kada je određena tačka podele, rekurzivno generišemo sve izraze čiji se listovi nalaze u levom delu vektora, sve izraze u kojima se listovi nalaze u desnom delu vektora, a zatim svaki izraz levo kombinujemo sa svakim izrazom desno, postavljajući svaki od četiri operatora u vrh.

```
// generiše sve moguće izraze kojima su vrednosti određene elementima niza
// vals na pozicijama [l, r]
vector<ExprPtr> generateExpressions(vector<int>& vals, int l, int r) {
  // niz vrednosti je jednočlan pa se jedino može napraviti list
  if (l == r)
    return vector<ExprPtr>(1, mkVal(vals[l]));
  // rezultujući vektor koji sadrži sve izraze
  vector<ExprPtr> result;
  // analiziramo sve moguće pozicije centralnog operatora
  for (int i = l; i < r; i++) {
    // rekurzivno generišemo sve leve i desne podizraze
    vector<ExprPtr> left, right;
    left = generateExpressions(vals, l, i);
    right = generateExpressions(vals, i+1, r);
    // kombinujemo ih na sve moguće načine, pomoću svih mogućih
    // operatora
    for (auto& el : left)
      for (auto& er : right) {
        vector<Op> ops{Add, Sub, Mul, Div};
        for (auto op : ops)
          result.push_back(mkApp(op, el, er));
```

```
      }
  }
  // vraćamo konačan rezultat
  return result;
}

// generiše sve moguće izraze kojima su vrednosti određene elementima niza
// vals na pozicijama [0, n)
vector<ExprPtr> generateExpressions(vector<int>& vals, int n) {
    return generateExpressions(vals, 0, n - 1);
}
```

Na kraju, ako želimo da odštampamo sve moguće izraze možemo upotrebiti naredni program.

```
int main() {
  vector<int> a{1, 3, 7, 10, 25, 50};
  subsets(a, [](vector<int>& a, int n) {
     permutations(a, n, [](vector<int>& a, int n) {
        for (ExprPtr e : generateExpressions(a, n)) {
            printExpr(e);
            cout << endl;
        }
    });
  });
  return 0;
}
```

Iscrpna pretraga (gruba sila)

n-dama - gruba sila

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. Važno pitanje je reprezentacija pozicija. Najbolje rešenje se dobija ako se rasporedi predstave permutacijama elemenata skupa $\{1, 2, \ldots, n\}$. 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;
  }
}
```

Provera da li je iskazna formula tautologija

Još jedan od problema koji ste tokom školovanja rešavali grubom silom je ispitivanje da li je formula tautologija. 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. Prikažimo jedno moguće rešenje. 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.

```
/* Struktura čvora drveta formule */

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

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

typedef shared_ptr<Cvor> CvorPtr;
```

```
/* Funkcije za kreiranje čvorova */

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;
}

/* Pomocne funkcije za lakše kreiranje čvorova */

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

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

Izračunavanje vrednosti formule je takođe veoma jednostavno.

```
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);
  }
}
```

Centralni deo zadatka predstavlja kombinatorna pretraga. Ona se zasniva

na algoritmu nabrajanja svih varijacija (u ovom slučaju koristimo pristup pronalaženja leksikografski sledeće varijacije).

```
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;
}
```

Da bismo znali koliko elemenata treba da ima valuacija, definisaćemo pomoćnu funkciju koja pronalazi najveću promenljivu koja se javlja u formuli.

```
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;
}
```

Funkcija za proveru tautologičnosti kreće od valuacije u kojoj su sve promenljive netačne, a koja ima isto onoliko elemenata koliko je promenljivih u formuli. Nakon toga se za svaku valuaciju proverava istinitosna vrednost formule sve dok se ne pojavi valuacija u kojoj je formula netačna ili dok se ne nabroje sve valuacije. Primetimo da se kod netautologičnih formula pretraga prekida i pre nego što se sve mogućnosti ispitaju (to ne utiče na vreme najgoreg slučaja, ali može značajno ubrzati neke konkretne slučajeve).

```
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;
}
```

Testiranje ovih funkcija u glavnom programu je jednostavno.

```
int main() {
  CvorPtr p = Prom(0);
```

```
  CvorPtr q = Prom(1);
  CvorPtr formula = Ekviv(Ne(Ili(p, q)), I(Ne(p), Ne(q)));
  cout << tautologija(formula) << endl;
  return 0;
}
```

Za rešenje ovog problema, naravno, postoje drugi, mnogo efikasniji algoritmi.

Slagalica

Problem: Odrediti sve izraze čija je vrednost jednaka datoj a u kojima učestvuju samo vrednosti iz datog niza celobrojnih vrednosti (svaka vrednost iz niza se može javiti samo jednom). Tokom izračunavanja izraza potrebno je sve vreme ostati u domenu prirodnih brojeva (nije dozvoljeno oduzimati veći broj od manjeg niti deliti brojeve koji nisu deljivi).

Rešenje zadataka grubom silom lako možemo da organizujemo oko proceduru generisanja svih izraza koju smo ranije definisali.

Potrebno je definisati funkcije za računanje vrednosti izraza i proveru da li su izrazi dobro definisani u odnosu na ograničeni domen operacija samo nad prirodnim brojevima.

```
// rezultat primene operatora na dva celobrojna operanda
int apply(Op op, int x, int y) {
  switch (op) {
  case Add:
    return x + y;
  case Sub:
    return x - y;
  case Mul:
    return x * y;
  case Div:
    return x / y;
  }
}

// provera da li se operator može primeniti u skupu prirodnih brojeva
bool valid(Op op, int x, int y) {
  if (op == Sub)
    return x > y;
  if (op == Div)
    return x % y == 0;
  return true;
}
```

Definišemo sada funkciju koja proverava da li se izraz može izračunati u skupu celih brojeva i ako može, koja mu je vrednost.

```
bool eval(ExprPtr e, int& value) {
  switch (e->type) {
  case Val:
    value = e->val;
    return true;
  case App:
    int x, y;
    if (!eval(e->left, x))
      return false;
    if (!eval(e->right, y))
      return false;
    if (!valid(e->op, x, y))
      return false;
    value = apply(e->op, x, y);
    return true;
  }
}
```

Na kraju možemo definisati funkciju koja generiše i proverava sve izraze i štampa one koji su korektni u skupu prirodnih brojeva i čija je vrednost jednaka datoj.

```
void generateAndCheckExpressions(vector<int>& vals, int n, int terget) {
  for (auto e : generateExpressions(vals, n)) {
    int val;
    if (eval(e, val) && val == target) {
      printExpr(e);
      cout << endl;
    }
  }
}
```

Glavnu funkciju možemo napisati na sledeći način.

```
int main() {
  vector<int> a{1, 3, 7, 10, 25, 50};
  int target = 765;
  subsets(a, [target](vector<int>& a, int n) {
      permutations(a, n, [target](vector<int>& a, int n) {
          generateAndCheckExpressions(a, n, target);
        });
    });
  return 0;
}
```

Naglasimo još jednom da je ovo rešenje rešenje grubom silom i da ima i efikasnijih načina da se zadatak reši.

Pretraga sa povratkom (bektreking)

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. Dakle, 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 (to utvrđujemo funkcijom `jestePotencijalnoResenje`, međutim, u realnim situacijama se taj uslov svede ili na to da je uvek tačan, što se dešava kada je svaki čvor drveta potencijalni kandidat za rešenje ili na to da je tačan samo u listovima, što se detektuje tako što se proveri da je `k` dostiglo maksimalnu vrednost). 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);
   }
}
```

Alternativno, provere umesto na ulazu u funkciju možemo vršiti pre rekurzivnih

poziva (čime se malo štedi na broju rekurzivnih poziva, ali se kod može zakomplikovati).

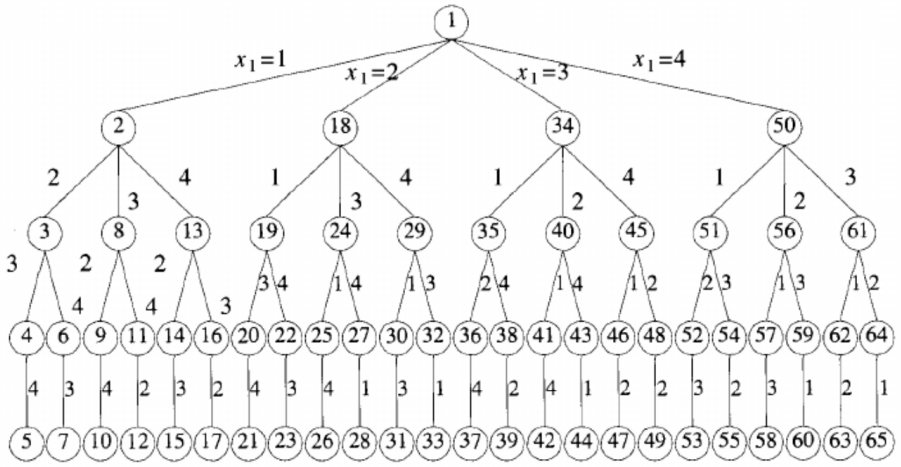
```
void pretraga(vector<int>& v, int k) {
   for (int x : kandidati(v, k)) {
      v[k] = x;
      if (!odsecanje(v, k)) {
         if (jestePotencijalnoResenje(v, k))
            ispisi(v, k);
         if (k < v.size())
            pretraga(v, k+1);
      }
   }
}
```

U ovoj situaciji treba obratiti pažnju i na to da prazan niz može predstavljati ispravno rešenje i da taj slučaj treba posebno obraditi pre prvog poziva ove funkcije.

Rekurzije je moguće osloboditi se uz korišćenje steka.

n-dama - pretraga sa povratkom

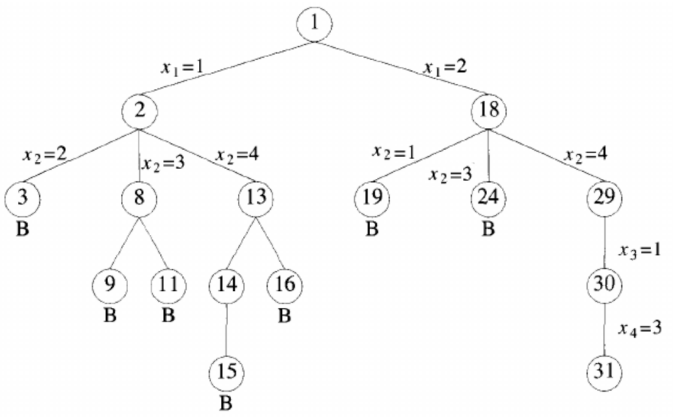
Prikažimo sada rešenje problema postavljanja n dama na šahovsku tablu, kog smo već rešavali grubom silom. Ceo prostor potencijalnih rešenja se može predstaviti drvetom u čijim se listovima nalaze permutacije koje predstavljaju rasporedi dama (ovo drvo je predstavljeno na slici 10). Rešenje grubom silom ispituje sve listove ovog drveta.



Slika 10: Drvo prostora rešenja problema 4 dame

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 slici 11. 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. Pošto ni jedan nastavak grane (13??) nije doveo do rešenja i tu granu odsecamo i postavljamo damu u drugoj vrsti u četvrtu kolonu (14??). Nastavljamo sa (142?), pa zatim i (1423), no tada se dame u poslednje dve vrste napadaju, pa se i ove grane odsecaju. Nakon toga prelazimo na granu 143? i odmah je odsecamo jer se grane u drugoj i trećoj vrsti napadaju. Pošto nema više mogućnosti za damu u drugoj vrsti, grana (1???) se odseca. Nakon toga prelazimo na (2???). Nastavci (21??) i (23??) se odmah odsecaju, jer se dame u prve dve vrste napadaju. Prelazimo na (24??), zatim na (241?) i na kraju dolazimo do (2413), što je prvo pronađeno uspešno rešenje.



Slika 11: Sasečeno drvo prostora rešenja problema 4 dame, do pronalaženja prvog rešenja

Pređimo sada na implementaciju. 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 (dodajući preostale dame u preostale kolone). 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 (ono je jedinstveno i ne može se proširiti). 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 (time zapravo umesto permutacija proveravamo varijacije, koje odmah sasecamo). Ako se ustanovi da to nije slučaj, tj. da se postavljanjem dame u tu vrstu ona ne napada ni sa jednom od prethodno postavljenih dama, 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.

```
// niz permutacija sadrži vrste dama u kolonama iz intervala [0, k]
// pretpostavlja se da se dame na pozicijama [0, k) ne napadaju
// i proverava se da li se dama na poziciji k napada sa nekom od njih
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;
}

// niz permutacija sadrži vrste dama u kolonama iz intervala [0, k) za
// koji se pretpostavlja da predstavlja raspored dama koje se ne napadaju
// procedura ispisuje sve moguće raspored dama koje proširuju taj raspored
// i u kojima se dame ne napadaju
void nDama(vector<int>& permutacija, int k) {
  // sve dame su postavljene
  if (k == permutacija.size())
    ispisi(permutacija);
  else {
```

```
    // postavljamo damu u kolonu k
    // isprobavamo sve moguće vrste
    for (int i = 0; i < permutacija.size(); i++) {
      // postavljamo damu u koloni k u vrstu i
      permutacija[k] = i;
      // proveravamo da li se dame i dalje ne napadaju
      if (!dameSeNapadaju(permutacija, k))
        // ako se ne napadaju, nastavljamo sa proširivanjem
        // tekućeg rasporeda
        nDama(permutacija, k+1);
    }
  }
}

void nDama(int n) {
   // niz koji za svaku kolonu beleži vrstu dame u toj koloni
   vector<int> permutacija(n);
   // krećemo pretragu od prazne table
   nDama(permutacija, 0);
}
```

Sudoku

Problem: Sudoku je zagonetka u kojoj se zahteva da se brojevi od 1 do 9 rasporede po polju dimenzije 9×9 tako da se u svakoj vrsti, svakoj koloni i svakom od 9 kvadrata dimenzije 3×3 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 $n^2 \times n^2$. 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 (uzevši u obzir i polja koja su zadata u početku). 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 (pošto na osnovu invarijante znamo da među popunjenim poljima nema konflikata). Ako to nije poslednje polje, samo se prelazi na popunjavanje narednog polja (konflikata i dalje nema, a polje možemo pomeriti na naredno jer znamo da će sva polja pre tog narednog biti popunjena, pa invarijanta ostaje održana). 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) {
  // da li se m[i][j] nalazi već u koloni j
  for (int k = 0; k < n * n; k++)
    if (k != i && m[i][j] == m[k][j])
      return true;

  // da li se m[i][j] nalazi već u vrsti i
  for (int k = 0; k < n * n; k++)
    if (k != j && m[i][j] == m[i][k])
      return true;

  // da li se m[i][j] već nalazi u kvadratu koji sadrži polje (i, j)
  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;

  // ne postoji konflikt
  return false;
}

// sudoku popunjavamo redom, vrstu po vrstu, kolonu po kolonu
// odredjuje polje koje treba popuniti nakon polja (i, j)
void sledeci(int& i, int& j) {
  j++;
  if (j == n * n) {
    j = 0;
    i++;
  }
}

bool sudoku(vector<vector<int>>& m, int i, int j) {
  // ako je polje (i, j) već popunjeno
  if (m[i][j] != 0) {
    // ako je u pitanju poslednje polje, uspešno smo popunili ceo
    // sudoku
    if (i == n * n - 1 && j == n * n - 1)
      return true;
    // prelazimo na naredno polje
    sledeci(i, j);
    // rekurzivno nastavljamo sa popunjavanjem
```

```
    return sudoku(m, i, j);
  } else {
    // razmatramo sve moguće vrednosti koje možemo da upišemo na polje
    // (i, j)
    for (int k = 1; k <= n*n; k++) {
      // upisujemo vrednost k
      m[i][j] = k;
      // ako time napravljen neki konflikt, nastavljamo popunjavanje
      // (pošto je polje popunjeno, na sledeće polje će se automatski
      // preći u rekurzivnom pozivu)
      // ako se sudoku uspešno popuni, prekidamo dalju pretragu
      if (!konflikt(m, i, j))
        if (sudoku(m, i, j))
          return true;
    }
    // poništavamo vrednost upisanu na polje (i, j)
    m[i][j] = 0;
    // konstatujemo da ne postoji rešenje
    return false;
  }
}
```

Među rešenjima postoji veliki broj ekvivalentnih, jer se mogu dobiti jedno od drugog simetrijama table. Ako se fokusiramo na horizontalnu simetriju, tada možemo pretpostaviti da se dama u prvom redu nalazi u levoj polovini table (ako nije, horizontalnom simetrijom možemo dobiti rešenje u kojem jeste). Ako je tabla neparne dimenzije i ako je dama u prvom redu tačno u srednjoj koloni, tada možemo pretpostaviti i da se dama u drugom redu nalazi na levoj polovini table (ako nije, opet se horizontalnom simetrijom može dovesti na nju). U tom slučaju nije moguće da se i u drugom redu dama nalazi u srednjoj koloni (jer bi se napadala sa onom u prvom redu).

Obilazak table skakačem

Problem: Napisati program koji pronalazi sve moguće načine da skakač u šahu obiđe celu šahovsku tablu, krenuvši iz gornjeg levog ugla, krećući se po pravilima šaha tako da na svako polje table stane tačno jednom. Dopustiti da dimenzija table bude različita od 8×8.

Zadatak se može rešiti pretragom sa povratkom, po uzoru na tehnike koje smo koristili prilikom generisanja kombinatornih objekata. Parcijalno rešenje u svakom koraku proširujemo tako što određujemo sve moguće načine da skakač pređe sa tekućeg na naredno polje. Za to je potrebno da znamo koje je polje već obiđeno, a koje je slobodno. Pošto ćemo želeti da rešenja prikažemo u nekom preglednom obliku, možemo održavati matricu u kojoj ćemo na neposećenim poljima čuvati nule, a na posećenim poljima redne brojeve poteza kada je to polje posećeno. Tokom obilaska održavaćemo i redni broj trenutnog poteza.

Definisaćemo rekurzivnu funkciju čiji je zadatak da proširi započeti obilazak time što će se naredni potez čiji je redni broj dat odigrati na polje čije su koordinate takođe date. Pretpostavićemo da će funkcija biti pozivana samo ako smo sigurni da taj potez može biti odigran. Nakon odigravanja poteza, proveravaćemo da li je tabla kompletno popunjena (to možemo jednostavno otkriti na osnovu dimenzija table i rednog broja odigranog poteza) i ako jeste, ispisivaćemo pronađeno rešenje. Ako nije, biraćemo naredni potez tako što analiziramo sve moguće poteze skakača, proveravati koji je moguće odigrati (koji vodi na neko nepopunjeno polje u okvirima table) i pozivaćemo rekurzivno funkciju da nastavi obilazak od tog polja.

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

using namespace std;

// dimenzije table - broj vrsta i broj kolona
const int V = 4, K = 5;

// svi mogući pomeraji prilikom poteza skakačem
vector<vector<int>> potezi =
  {{-1, -2}, {-1, 2}, {1, -2}, {1, 2},
   {-2, -1}, {-2, 1}, {2, -1}, {2, 1}};

// ispis table sa rednim brojevima poteza na standardni izlaz
void ispisi(const vector<vector<int>>& tabla) {
  for (int v = 0; v < V; v++) {
    for (int k = 0; k < K; k++)
      cout << setw(2) << tabla[v][k] << " ";
    cout << endl;
  }
  cout << endl;
}

// centralna funkcija koja vrši pretragu sa povratkom
void obidjiTabluSkakacem(vector<vector<int>>& tabla,
                         int v, int k, int potez) {
  tabla[v][k] = potez;
  if (potez == V*K)
    ispisi(tabla);
  for (auto& p : potezi) {
    int vn = v + p[0], kn = k + p[1];
    if (0 <= vn && vn < V &&
        0 <= kn && kn < K &&
        tabla[vn][kn] == 0) {
      obidjiTabluSkakacem(tabla, vn, kn, potez+1);
    }
  }
```

```
  tabla[v][k] = 0;
}

int main() {
  // alociramo tablu u koju upisujemo slobodna polja
  // i redne brojeve poteza
  vector<vector<int>> tabla(V);
  for (int i = 0; i < V; i++)
    tabla[i].resize(K, 0);
  // pokrećemo pretragu od prvog poteza u gornjem levom uglu
  obidjiTabluSkakacem(tabla, 0, 0, 1);

  return 0;
}
```

Primetimo da je na kraju funkcije potrebno eksplicitno da poništimo odluku, tj. da polje (v, k) ekplicitno označimo kao slobodno.

Izlazak iz lavirinta

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).

```
xxxxxxxxxxxxxxx
xS...........x
x.xxxxxxxxxxxx
x............x
xxxxxxxxxxxx.x
x............x
x.xxxxxxx.xxxxx
x...........Cx
xxxxxxxxxxxxxxx
```

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.

```
// ispituje da li postoji put kroz lavirint od polja (x1, y1) do (x2, y2)
// pri čemu matrica posecen označava koja su polja posećena, a koja nisu
bool pronadjiPut(Matrica<bool>& lavirint, Matrica<bool>& posecen,
                 int x1, int y1, int x2, int y2) {
  // ako se početno i krajnje polje poklapaju, put postoji
  if (x1 == x2 && y1 == y2)
    return true;
  // četiri smera u kojima se možemo pomerati
  int pomeraj[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
  // pokušavamo pomeranja u svakom od tih smerova
  for (int i = 0; i < 4; i++) {
    // polje na koje se pomeramo
    int x = x1 + pomeraj[i][0], y = y1 + pravac[i][1];
    // ako tu nije zid i ako to polje nije posećeno
    if (!lavirint[x][y] && !posecen[x][y]) {
      // pomeramo se na to polje, beležimo da je posećeno
      posecen[x][y] = true;
      // ako od njega postoji put, onda postoji put i od polja (x1, y1)
      if (pronadjiPut(lavirint, posecen, x, y, x2, y2))
        return true;
    }
  }
  // ni u jednom od 4 moguća smera nismo uspeli da stignemo do cilja,
  // pa put ne postoji
  return false;
}


// ispituje da li postoji put kroz lavirint od polja (x1, y1) do (x2, y2)
bool pronadjiPut(Matrica<bool>& lavirint,
                 int x1, int y1, int x2, int y2) {
  // dimenzije lavirinta
  int m = lavirint.size(), n = lavirint[0].size();
  // ni jedno polje osim starta do sada nije posećeno
  Matrica<bool> posecen = napraviMatricu(m, n, false);
  posecen[x1][y1] = true;
  // pokrećemo rekurzivnu pretragu od startnog polja
  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 (proveriti da li se na njima nalazi cilj) 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 (npr. -1 i po tome ih raspoznavati). 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.

```
// ispituje da li postoji put kroz lavirint od polja (x1, y1) do (x2, y2)
int najkraciPut(Matrica<bool>& lavirint,
                int x1, int y1, int x2, int y2) {
  // dimenzije lavirinta
  int m = lavirint.size(), n = lavirint[0].size();
  // matrica u kojoj se beleži najkraće rastojanje od polja (x1, y1)
  // -1 označava da to rastojanje još nije određeno
  Matrica<int> rastojanje = napraviMatricu(m, n, -1);
  // krećemo od polja (x1, y1) koje je samo od sebe udaljeno 0 koraka
  rastojanje[x1][y1] = 0;
  // red polja koje treba obraditi - polja se u red dodaju po neopadajućem
  // rastojanju od (x1, y1)
  queue<pair<int, int>> red;
  // pretraga kreće od (x1, y1)
  red.push(make_pair(x1, y1));
  // dok se red ne isprazni
  while (!red.empty()) {
    // skidamo element iz reda
    x1 = red.front().first; y1 = red.front().second;
    red.pop();
    // čitamo rastojanje od (x1, y1) do njega
    int r = rastojanje[x1][y1];
    // ako je to ciljno polje, odredili smo najkraće rastojanje
    if (x1 == x2 && y1 == y2)
      return r;
    // četiri smera u kojima se možemo pomerati
    int pomeraj[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    // pokušavamo pomeranja u svakom od tih smerova
    for (int i = 0; i < 4; i++) {
      // polje na koje se pomeramo
      int x = x1 + pomeraj[i][0], y = y1 + pomeraj[i][1];
      // ako tu nije zid i ako to polje nije ranije posećeno
      if (!lavirint[x][y] && rastojanje[x][y] == -1) {
        // odredili smo najkraće rastojanje od (x1, y1) do njega
        rastojanje[x][y] = r + 1;
        // postavljamo ga u red za dalju obradu
        red.push(make_pair(x, y));
      }
    }
  }
  return -1;
```

```
}
```

Slagalica - pretraga sa povratkom

Mana rešenja slagalice zasnovanog na gruboj sili je to što je veliki broj generisanih izraza neispravan kada se dopuste samo operacije u skupu prirodnih brojeva. Provera ispravnosti izraza vrši se tek nakon generisanja celog izraza. Pošto ispravan izraz ne može imati neispravan podizraz, velike uštede se mogu dobiti ako se provera ispravnosti izraza vrši tokom samog generisanja izraza i ako se u svakom koraku generisanja odbace svi (pod)izrazi za koje se ustanovi da nisu ispravni nad prirodnim brojevima.

```
// generiše sve moguće izraze kojima su vrednosti određene elementima
// niza vals na pozicijama [l, r]
vector<pair<ExprPtr, int>>
generateExpressionsAndValues(vector<int>& vals, int l, int r) {
  // niz vrednosti je jednočlan pa se jedino može napraviti list
  if (l == r) {
    auto p = make_pair(mkVal(vals[l]), vals[l]);
    return vector<pair<ExprPtr, int>>(1, p);
  }
  // rezultujući vektor koji sadrži sve izraze
  vector<pair<ExprPtr, int>> result;
  // analiziramo sve moguće pozicije centralnog operatora
  for (int i = l; i < r; i++) {
    // rekurzivno generišemo sve leve i desne podizraze
    vector<pair<ExprPtr, int>> left, right;
    left = generateExpressionsAndValues(vals, l, i);
    right = generateExpressionsAndValues(vals, i+1, r);
    // kombinujemo ih na sve moguće načine, pomoću svih mogućih
    // operatora
    for (auto& el : left)
      for (auto& er : right) {
        vector<Op> ops{Add, Sub, Mul, Div};
        for (auto op : ops) {
          if (valid(op, el.second, er.second)) {
            auto p = make_pair(mkApp(op, el.first, er.first),
                               applyOp(op, el.second, er.second));
            result.push_back(p);
          }
        }
      }
  }
  // vraćamo konačan rezultat
  return result;
}

// pronalazimo i ispisujemo sve ispravne izraze čija je vrednost
```

```
// jednaka ciljnoj
void checkExpressions(vector<int>& vals, int n, int target) {
  vector<pair<ExprPtr, int>> expressionsAndValues =
    generateExpressionsAndValues(vals, 0, n - 1);
  for (auto e : expressionsAndValues)
    if (e.second == target) {
      printExpr(e.first);
      cout << endl;
    }
}
```

Ovim se vreme generisanja značajno smanjuje. Analiza dobijenih rezultata pokazuje da među izrazima ima veliki broj njih koji su međusobno ekvivalentni, do na komutativnost i asocijativnost operatora, kao i da su neki nepotrebno komplikovani jer se u njima koriste neutralne vrednosti. Komutativnost i neutrale je moguće veoma jednostavno ukloniti, ako malo pojačamo funkciju kojom se proverava ispravnost izraza. Dodatni uslov koji namećemo je taj da će kod sabiranja i množenja prvi operand uvek biti manji ili jednak od drugog.

```
bool valid(Op op, int x, int y) {
  if (op == Add)
    return x <= y;
  if (op == Sub)
    return x > y;
  if (op == Mul)
    return x <= y && x != 1 && y != 1;
  if (op == Div)
    return x % y == 0 && y != 1;
  return true;
}
```

I ovim se vreme generisanja značajno smanjuje, ali po cenu generisanja mnogo manjeg broja izraza. Među dobijenim izrazima ostaju oni koji su ekvivalentni do na asocijativnost i njih nije tako jednostavno ukloniti.

Podskup elemenata sa datim zbirom

Problem: Dat je skup pozitivnih brojeva. Napisati program koji određuje koliko elemenata tog skupa ima zbir jednak datom pozitivnom broju.

Pretpostavićemo da je skup zadat nizom različitih elemenata, međutim, svi algoritmi koje navodimo radili bi ispravno i ako bi u nizu bilo ponavljanja i ako bi se zahtevalo da se prebroje svi podnizovi (ne obavezno susednih elemenata) čiji je zbir jednak datom 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 (kada bi bilo, mogli bismo održavati vektor logičkih vrednosti kojim se predstavlja koji su elementi uključeni u podskup, a koji nisu). 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.

```
// funkcija određuje broj podskupova skupa određenog elementima na
// pozicijama [k, n) takvih da je zbir elemenata podskupa jednak
// ciljnom zbiru
int brojPodskupovaDatogZbira(const vector<int>& skup,
                             int ciljniZbir, int k) {
  // jedino prazan skup ima zbir nula
  if (ciljniZbir == 0)
    return 1;

// jedini podskup praznog skupa je prazan, a ciljni zbir je pozitivan
  if (k == skup.size())
    return 0;

  // posebno brojimo podskupove koji uključuju i one koji ne uključuju
  // element skup[k]
  return brojPodskupovaDatogZbira(skup, ciljniZbir - skup[k], k+1) +
         brojPodskupovaDatogZbira(skup, ciljniZbir, k+1);
}

int brojPodskupovaDatogZbira(const vector<int>& skup, int ciljniZbir) {
  // brojimo podskupove skupa određenog elementima na pozicijama [0, n)
  return  brojPodskupovaDatogZbira(skup, ciljniZbir, 0);
}
```

Međutim, 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. Time dobijamo narednu implementaciju.

```
// funkcija računa broj podskupova elemenata skupa na pozicijama [k, n)
// koji imaju dati zbir, pri čemu se zna da je zbir tih elemenata jednak
// broju zbirPreostalih
int brojPodskupovaDatogZbira(const vector<int>& skup, int ciljniZbir,
                             int zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan element
  if (ciljniZbir == 0)
    return 1;

  // jedini podskup praznog skupa je prazan, a ciljni zbir je pozitivan
  if (k == skup.size())
    return 0;

  // pošto su svi brojevi pozitivni, nije moguće dobiti negativan
  // ciljni zbir
  if (ciljniZbir < 0)
    return 0;

  // čak ni uzimanje svih elemenata ne može dovesti do ciljnog zbira,
  // pa nema podskupova koji bi dali ciljni zbir
  if (zbirPreostalih < ciljniZbir)
    return 0;

        // broj podskupova u kojima učestvuje element a[k]
  return brojPodskupovaDatogZbira(skup, ciljniZbir - skup[k],
                                  zbirPreostalih - skup[k], k+1) +
        // broj podskupova u kojima ne učestvuje element a[k]
        brojPodskupovaDatogZbira(skup, ciljniZbir,
                                  zbirPreostalih - skup[k], k+1);
}

// funkcija računa koliko podskupova datog skupa ima zbir jednak ciljnom
int brojPodskupovaDatogZbira(vector<int>& skup, int ciljniZbir) {
  // broj elemenata skupa
  int n = skup.size();
  // izračunavamo zbir elemenata skupa
  int zbirSkupa = 0;
  for (int i = 0; i < n; i++)
    zbirSkupa += skup[i];
  // rekurzivnom funkcijom računamo traženi broj podskupova
  return  brojPodskupovaDatogZbira(skup, ciljniZbir, zbirSkupa, 0);
}
```

Zbir podskupa može biti nula samo u slučaju praznog skupa. Ako skup nije prazan, tada donju granicu možemo popraviti tako što primetimo da je zbir podskupa uvek veći ili jednak njegovom najmanjem elementu. Ako je najmanji element skupa strogo veći od ciljnog zbira, tada odmah znamo da se ciljni zbir

ne može postići. Postavlja se pitanje kako efikasno inkrementalno da računamo najmanji element skupa. 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 (u svakom pozivu u kome je niz dužine n sortiran i u kome je prvi element strogo veći od ciljnog zbira uštedelo bi se $2n$ rekurzivnih poziva).

```
// funkcija računa broj podskupova elemenata skupa na pozicijama [k, n) koji
// imaju dati zbir, pri čemu se zna da je zbir tih elemenata jednak
// broju zbirPreostalih
int brojPodskupovaDatogZbira(const vector<int>& skup, int ciljniZbir,
                             int zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan element
  if (ciljniZbir == 0)
    return 1;
  // čak ni uzimanje svih elemenata ne može dovesti do ciljnog zbira,
  // pa nema podskupova koji bi dali ciljni zbir
  if (zbirPreostalih < ciljniZbir)
    return 0;
  // već uzimanje najmanjeg elementa prevazilazi ciljni zbir, pa
  // nema podskupova koji bi dali ciljni zbir
  if (skup[k] > ciljniZbir)
    return 0;

        // broj podskupova u kojima učestvuje element a[k]
  return brojPodskupovaDatogZbira(skup, ciljniZbir - skup[k],
                                  zbirPreostalih - skup[k], k+1) +
        // broj podskupova u kojima ne učestvuje element a[k]
        brojPodskupovaDatogZbira(skup, ciljniZbir,
                                  zbirPreostalih - skup[k], k+1);
}

// funkcija računa koliko podskupova datog skupa ima zbir jednak ciljnom
int brojPodskupovaDatogZbira(vector<int>& skup, int ciljniZbir) {
  // broj elemenata skupa
  int n = skup.size();
  // sortiramo elemente skupa neopadajuće
  sort(begin(skup), end(skup));
  // izračunavamo zbir elemenata skupa
  int zbirSkupa = accumulate(begin(skup), end(skup), 0);
  // rekurzivnom funkcijom računamo traženi broj podskupova
  return  brojPodskupovaDatogZbira(skup, ciljniZbir, zbirSkupa, 0);
}
```

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.

Ovaj zadatak je donekle sličan prethodnom. Osnovu čini rekurzivna provera svih podskupova. Upotrebićemo ovaj zadatak da ilustrujemo još jedan tip odsecanja. 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.

```
// funkcija određuje najmanju razliku između ciljne mase koju treba
// izmeriti i mase koja se može postići pomoću tegova iz intervala
// pozicija [k, n) čija je ukupna masa jednaka broju preostalaMasa
double merenje(const vector<double>& tegovi, double ciljnaMasa,
               int k, double tekucaMasa, double preostalaMasa) {
  // nemamo više tegova koje možemo uzimati
  if (k == tegovi.size())
    // najmanja razlika je određena tekućom masom uključenih tegova
    return abs(ciljnaMasa - tekucaMasa);

  // preskačemo teg na poziciji k
  double minRazlika = merenje(tegovi, ciljnaMasa, k+1,
                              tekucaMasa, preostalaMasa - tegovi[k]);
  // ako uključivanje tega na poziciji k ima šanse da da manju razliku
  // od najmanje
  if (tekucaMasa + preostalaMasa > ciljnaMasa - minRazlika &&
      tekucaMasa + tegovi[k] < ciljnaMasa + minRazlika) {
    // uključujemo ga i gledamo da li je to popravilo najmanju razliku
    double razlika = merenje(tegovi, ciljnaMasa, k+1,
                             tekucaMasa + tegovi[k], preostalaMasa - tegovi[k]));
    if (razlika < minRazlika)
       minRazlika = razlika;
  }
  // vraćamo najmanju razliku
  return minRazlika;
}

double merenje(const vector<double>& tegovi, double ciljnaMasa) {
  // određujemo ukupnu masu svih tegova
  double ukupnaMasa = 0.0;
  for (int i = 0; i < tegovi.size(); i++)
    ukupnaMasa += tegovi[i];
  // krećemo od pozicije 0 i praznog skupa uključenih tegova
  return merenje(tegovi, ciljnaMasa, 0, 0.0, ukupnaMasa);
}
```

Bratska podela na minimalnu razliku

Problem: Dva brata žele da podele predmete koje imaju u kući, tako da podela bude što pravednija, tj. da apsolutna razlika između vrednosti svih predmeta koje su dobili bude minimalna. Napisati program koji određuje vrednost te razlike.

Rešenje grubom silom podrazumeva da se isprobaju svi podskupovi skupa kao mogućnosti za skup predmeta prvog brata (njihovi komplementi onda odgovaraju skupovima predmeta drugog brata i da se pronađe najmanja razlika).

```
// računa apsolutnu vrednost razlike između predmeta koje su braća
// dobila, ako su predmeti prvog brata zadati vektorom predmetiPrvog
double razlika(const vector<double>& predmeti,
               const vector<bool>& predmetiPrvog) {
  // broj predmeta
  int n = predmeti.size();
  // zbir vrednosti predmeta prvog i drugog brata
  double prvi = 0.0, drugi = 0.0;
  // delimo predmete među braćom
  for (int i = 0; i < n; i++)
    if (predmetiPrvog[i])
      prvi += predmeti[i];
    else
      drugi += predmeti[i];
  // izračunavamo i vraćamo apsolutnu vrednost razlike
  return abs(prvi - drugi);
}

// funkcija izračunava najmanju apslutnu vrednost razlike između
// braće nakon podele predmeta
double podela(const vector<double>& predmeti) {
  // broj predmeta
  int n = predmeti.size();
  // skup predmeta koji pripadaju prvom bratu
  vector<bool> predmetiPrvog(n, false);
  // najmanju razliku inicijalizujemo na +beskonačno
  double minRazlika = numeric_limits<double>::max();
  do {
    // određujemo razliku za tekući skup
    double r = razlika(predmeti, predmetiPrvog);
    // ažuriramo minimum ako je potrebno
    if (r < minRazlika)
      minRazlika = r;
    // prelazimo na naredni podskup dok god je to moguće
  } while (sledeciPodskup(predmetiPrvog));
  // vraćamo konačnu minimalnu vrednost razlike
  return minRazlika;
}
```

Problem sa ovim pristupom je što se zbirovi predmeta računaju svaki put iznova. Bolje rešenje dobijamo ako podskupove nabrajamo rekurzivno i ako zbirove prvog i drugog brata inkrementalno računamo tokom rekurzije.

```
// funkcija određuje najmanju moguću vrednost razlike između
// braće nakon podele predmeta na pozicijama [0, n-1], ako prvi
// brat već ima predmete u vrednosti zbirPrvog, a drugi već ima predmete
// u vrednosti zbirDrugog
double podela(const vector<double>& predmeti, int n,
              double zbirPrvog, double zbirDrugog) {
  // nema više predmeta za podelu
  if (n == 0)
    return abs(zbirPrvog - zbirDrugog);

  // predmet na poziciji n-1 dajemo prvom, pa onda drugom bratu i gledamo
  // bolju od te dve mogućnosti
  return min(podela(predmeti, n-1, zbirPrvog + predmeti[n-1], zbirDrugog),
             podela(predmeti, n-1, zbirPrvog, zbirDrugog + predmeti[n-1]));
}

// funkcija izračunava najmanju apslutnu vrednost razlike između
// braće nakon podele predmeta
double podela(const vector<double>& predmeti) {
  // broj predmeta
  int n = predmeti.size();
  // delimo predmete na pozicijama [0, n) i braća nemaju ništa u startu
  return podela(predmeti, n, 0.0, 0.0);
}
```

Jedno jednostavno poboljšanje ove procedure se zasniva na simetriji odnosno na činjenici da možemo pretpostaviti da će prvi predmet pripasti prvom bratu (jer u suprotnom braća mogu da zamene svoje predmete i apsolutna razlika njihovih vrednosti se neće promeniti).

```
double podela(const vector<double>& predmeti) {
  // broj predmeta
  int n = predmeti.size();
  // delimo predmete na pozicijama [0, n-1), a prvom bratu dajemo
  // predmet na poziciji n-1
  return podela(predmeti, n-1, predmeti[n-1], 0.0);
}
```

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.

```
// funkcija boji dati cvor, pri cemu su zadati
// susedi svih cvorova i boje do sada obojenih cvorova
bool oboj(const vector<vector<int>>& susedi,
          int cvor, vector<int>& boje) {
  // ako su svi cvorovi obojeni, bojenje sa 3 boje je uspesno
  int brojCvorova = susedi.size();
  if (cvor >= brojCvorova)
    return true;

  // pokusavamo da cvoru dodelimo svaku od 3 raspolozive boje
  for (int boja = 1; boja <= 3; boja++) {
    // linearnom pretragom proveravamo da li je moguce obojiti cvor
    // u tekucu boju
    bool mozeBoja = true;
    // proveravamo sve susede
    for (int sused : susedi[cvor])
      // ako je neki od njih vec obojen u tekucu boju
      if (boje[sused] == boja)
        // bojenje nije moguce
        mozeBoja = false;
    if (mozeBoja) {
      // bojimo tekuci cvor
      boje[cvor] = boja;
      // pokusavamo rekurzivno bojenje narednog cvora i ako uspemo
      // tada je bojenje moguce
      if (oboj(susedi, cvor+1, boje))
        return true;
    }
  }

  // probali smo sve tri boje i ni jedno bojenje nije moguce
  return false;
}

bool oboj(const vector<vector<int>>& susedi, vector<int>& boje) {
  // broj cvorova grafa
  int brojCvorova = susedi.size();
  // sve pocetne nepovezane cvorove bojimo u proizvoljnu boju (npr. 1)
  int cvor = 0;
  while (cvor < susedi.size() && susedi[cvor].size() == 0)
    boje[cvor] = 1;
  // prva dva povezana cvora se boje u boje 1 i 2
  boje[cvor] = 1; boje[susedi[0][0]] = 2;
  // rekurzivno pokusavamo bojenje narednih cvorova
  return oboj(susedi, cvor, boje);
}
```

Grananje i odsecanje

k-bojenje grafa

0-1 problem ranca

Čas 11.1, 11.2, 11.3 - dinamičko programiranje

Pojam i oblici dinamičkog programiranja

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

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

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

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

Fibonačijevi brojevi

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

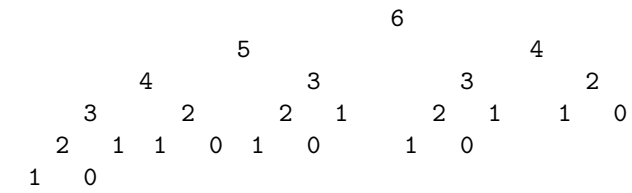
Pošto je već sama definicija rekurzivna, funkcija se može implementirati krajnje jednostavno.

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
int fib(int n) {
   int pp = 0;
   if (n == 0) return pp;
   int p = 1;
   if (n == 1) return p;
```

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

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

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

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

Prebrojavanje kombinatornih objekata

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

Broj kombinacija

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

Jedna interesantna tehnika koju smo već sreli prilikom empirijske analize složenosti funkcije QuickSort prilagođava algoritam tako da izračunava broj

koraka koji se u njemu izvršavaju. Tako možemo krenuti od funkcije koju smo izveli za generisanje svih kombinacija.

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

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }

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

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

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

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

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

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

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

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

Primećujemo da smo ovom transformacijom dobili čuvene osobine binomnih koeficijenata.

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

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

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

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

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

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

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

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

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

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

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

```
    for (int k = 1; k < n; k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // na kraju svake vrste nalazi se 1
    dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}
```

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

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

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

se k smanjuje za 1, čime se invarijanta održava. Ažuriranje se vrši do pozicije $k = 1$, jer se na poziciji $k = 0$ u svim vrstama nalazi vrednost 1.

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

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

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

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

Broj kombinacija sa ponavljanjem

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

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

Broj ovih kombinacija može se rasporediti u pravougaonik.

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

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

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

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

Broj particija

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

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

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

- Bazu indukcije čini slučaj $n = 0$, jer broj nula ima samo jednu particiju koja ne sadrži sabirke. Dakle, važi da je $p_{0,s} = 1$. Ako je n veće od nula i $s = 0$, tada ne postoji ni jedna particija, jer od sabiraka koji su svi jednaki nuli (jer svi moraju da budu manji ili jednaki s) ne možemo nikako napraviti neki pozitivan broj. Dakle, za $n > 0$ važi da je $p_{n,0} = 0$.
- Induktivni korak možemo ostvariti na više načina. Najjednostavniji je sledeći. Prilikom izračunavanja $p_{n,s}$ možemo razmatrati dva slučaja: da se u zbiru ne javlja sabirak s ili da se u zbiru javlja sabirak s. Ako se u zbiru ne javlja sabirak s, tada je najveći sabirak $s-1$ i broj takvih particija je $p_{n,s-1}$. Drugi slučaj je mogući samo kada je $n \geq s$ i broj takvih particija je $p_{n-s,s}$. Na osnovu ovoga, dobijamo narednu rekurzivnu definiciju.

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

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

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

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

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

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

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

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

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

```
int brojParticija(int N) {
  // alociramo matricu
  vector<vector<int>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(N+1);
  // popunjavamo prvu vrstu
  for (int smax = 0; smax <= N; smax++)
```

```
    dp[0][smax] = 1;
  // popunjavamo preostale elemente prve kolone
  for (int n = 1; n <= N; n++)
    dp[n][0] = 0;
  // popunjavamo jednu po jednu vrstu
  for (int n = 1; n <= N; n++)
    for (int smax = 1; smax <= N; smax++) {
      dp[n][smax] = dp[n][smax-1];
      if (n >= smax)
        dp[n][smax] += dp[n-smax][smax];
    }
  return dp[N][N];
}
```

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

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

Broj načina dekodiranja

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

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

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

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

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

U ovom rekurzivnom rešenju (bez ozbira na način implementacije) postojaće

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

Jedan od načina da smanjimo broj rekurzivnih poziva i nepotrebna izračunavanja je memoizacija. U pomoćnom nizu pamtimo do sada izračunate rezultate (na poziciji i pamtimo vrednost rekurzivnog poziva za parametar i). Niz inicijalizujemo na sve vrednosti -1. Na početku rekurzivne funkcije proveravamo da li je vrednost na poziciji i različita od -1 i ako jeste, vraćamo je. U suprotnom nastavljamo izvršavanje rekurzivne funkcije. Pre nego što funkcija vrati rezultat upisujemo ga u niz na mesto i.

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

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

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

Možemo se osloboditi rekurzije i niz popuniti odozda naviše.

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

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

U suprotnom u nizu imamo bar jednu cifru.

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

Ako je s_i nije cifra `'0'`, treba da razmotrimo mogućnost dekodiranja na prvi i na drugi način. Prvi način je uvek primenljiv (jer sufiks počinje bar jednom cifrom s_i za koju smo utvrdili da nije cifra `'0'`) i u tom slučaju niz možemo dekodirati na dp_{i+1} načina. Drugi način je primenljiv ako imamo bar dve cifre (ako je $i < n-1$) i ako je broj dobijen od cifara $s_i s_{i+1}$ (za s_i smo utvrdili da nije cifra `'0'`) između 1 i 26 i u tom slučaju je broj načina dobijen na ovaj način jednak dp_{i+2}. Jednostavnosti radi dp_{n-1} je moguće odrediti posebno, da se ne bi stalno proveravalo da li je $i < n-1$.

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

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

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

```
long long brojNacinaDekodiranja(const string& s) {
  int n = s.length();
  // krećemo od dp[n] = 1
  long long dp2 = 1;
  // i od dp[n-1] što je 1 ako s[n-1] nije karakter '0' i 0 ako jeste
  long long dp1 = s[n-1] != '0';
  // niz dp rekonstruišemo unatrag
  for (int i = n-2; i >= 0; i--) {
    // vrednost dp[i]
    long long dp = 0;
    // prvu cifru dekodiramo posebno (ako čini broj između 1 i 9)
    if (s[i] == '0')
       dp += dp1;
    // prve dve cifre dekodiramo zajedno (ako čine broj između 10 i 26)
    if (10 * (s[i]-'0') + (s[i+1]-'0') <= 26)
       dp += dp2;
    // ažuriramo dva poslednja poznata člana niza
    dp2 = dp1;
    dp1 = dp;
  }
  return dp1;
}
```

Broj sečenja n-tougla na trouglove

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

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

Na osnovu ovoga je jednostavno dati rekurzivnu formulaciju.

```
int katalan(int n) {
```

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

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

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

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

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

Zbir podskupa (varijanta 0-1 ranca)

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

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

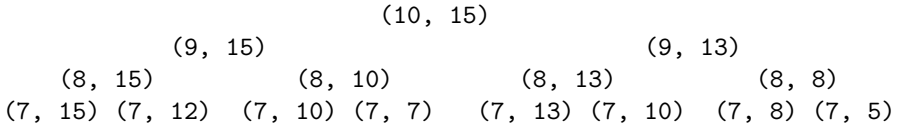
Rešenje grubom silom bi podrazumevalo da se isprobaju svi podskupovi predmeta. Složenost tog pristupa bi bila $O(2^n)$. Naglasimo da smo ovaj prisup već implementirali u delu pretrazi, no sada ćemo ga unaprediti, pod pretpostavkom da su nosivost ranca i mase predmeta relativno mali prirodni brojevi.

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

```
bool zbirPodskupa(vector<int>& m, int n, int M) {
   // preostala nosivost je 0, pa je ranac kompletno popunjen
   if (M == 0)
      return true;
   // preostala nosivost je pozitivna, ali nemamo više predmeta
   // pa je ne možemo popuniti
   if (n == 0)
      return false;
   // preskačemo n-ti predmet i pokušavamo da ranac napunimo sa
   // prvih n-1 predmeta
   if (zbirPodskupa(m, n-1, M))
      return true;
   // ako n-ti predmet može da stane u ranac stavljamo ga i
   // pokušavamo da preostalu nosivost popunimo pomoću preostalih
   // n-1 predmeta
   if (m[n-1] <= M && zbirPodskupa(m, n-1, M - m[n-1]))
      return true;
   // ako ne možemo da napravimo podskup ni sa ni bez n-tog predmeta
   // onda podskup nije moguće napraviti
   return false;
}


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

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

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



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

ponovljeno. Ubrzanje se stoga vrlo verovatno može dobiti dinamičkim programiranjem.

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

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

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

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

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

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

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

  for (int n = 1; n <= N; n++) {
```

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

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

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

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

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

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

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

Implementacija postupka rekonstrukcije rešenja može biti sledeća.

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

```
    m -= masa[n];
  }
  return podskup;
}
```

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

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

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

  return dp[M];
}
```

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

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

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

  return dp[M];
}
```

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

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

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

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

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

Rešavanje optimizacionih problema dinamičkim programiranjem

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

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

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

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

Minimalni broj novčića

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

Problem: Na raspolaganju imamo n novčića čiji su iznosi celi brojevi $m_0, \ldots, m_{n-1}$. Napisati program koji određuje minimalni broj novčića pomoću kojih se može platiti dati iznos (svaki novčić se može upotrebiti samo jednom).

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

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

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

formulaciju.

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

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

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

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

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

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

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

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

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

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

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

Naravno, moguće je i rešenje odozdo naviše.

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

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

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

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

Kada je izračunata matrica, rekonstrukciju rešenja možemo izvršiti veoma jednostavno. Krećemo iz donjeg desnog ugla i u svakom trenutku proveravamo da li je broj jednak onom iznad sebe ili onom elementu prethodne vrste koji se dobija umanjivanjem tekućeg iznosa za vrednost tekućeg novčića (moguće je i da je jednak obema tim vrednostima i tada su moguća različita rešenja, pa se možemo opredeliti za bilo koje od njih). U prethodnom primeru su označene vrednosti koje se koriste tokom rekonstrukcije. Iznos 15 se gradi od 3 novčića iz celog skupa. Pošto se iznad broja 3 ne nalazi trojka moramo uzeti novčić 6 i prelazimo na rekonstrukciju iznosa 9 pomoću 2 novčića iz skupa koji sadrži prva 4 novčića. Iznos 9 je moguće rekonstruisati pomoću 2 novčića iz skupa koji sadrži prva 3 novčića, a iznos 5 nije moguće rekonstruisati pomoću jednog

novčića iz tog skupa, što znači da novčić 4 moramo preskočiti. Pošto iznos 9 nije moguće rekonstruisati samo pomoću prva dva novčića, nović 2 moramo uzeti i nakon toga rekonstruišemo iznos 7 pomoću prva dva novčića. Pošto 7 nije moguće formirati samo od prvog novčića, moramo uzeti novčić 7, nakon čega stižemo do iznosa 0. Time se rekonstrukcija završava i odabrali smo novčiće $7 + 2 + 6 = 15$.

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

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

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

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

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

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

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

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

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

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

  int zbir = 0;
```

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

Maksimalni zbir segmenta

Razmotrimo problem određivanja maksimalnog zbira segmenta iz ugla dinamičkog programiranja.

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

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

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

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

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

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

```
int maksimalniZbirSegmenta(const vector<int>& a) {
   int n = a.size();
   // za svaku poziciju određujemo maksimalni zbir sufiksa
   // koji se završava na toj poziciji
   vector<int> dp(n + 1);
   dp[0] = 0;
   for (int i = 1; i <= n; i++)
      dp[i] = max(0, dp[i-1] + a[i-1]);
   // određujemo maksimalni zbir segmenta (on je sigurno zbir
   // nekog sufiksa, pa tražimo maksimum svih zbirova sufiksa)
   int rez = dp[0];
   for (int i = 1; i <= n; i++)
      rez = max(rez, dp[i]);
   return rez;
}
```

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

Naravno, dva prolaska možemo objediniti u jedan.

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

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

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

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

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

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

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

Najduži zajednički podniz

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

Krećemo od rešenja induktivno-rekurzivnom konstrukcijom.

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

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

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

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

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

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

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

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

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

```
  return memo[n1][n2] = rez;
}

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

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

Problem preklpajućih rekurzivnih poziva se može rešiti ako se upotrebi dinamičko programiranje naviše. Dužine najdužih podnizova prefiksa možemo čuvati u matrici. Element matrice na poziciji (m, n) zavisi samo od elemenata na pozicijama $(m-1, n)$, $(m, n-1)$ i $(m-1, n-1)$, tako da matricu požemo da popunjavamo bilo vrstu po vrstu, bilo kolonu po kolonu. Prikažimo matricu za primer niske `xmjyauz` i `mzjawxu`.

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

Implementaciju možemo izvršiti na sledeći način.

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

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

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

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

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

  string rez;
  rez.resize(dp[n1][n2]);
  int i = n1, j = n2, k = dp[n1][n2] - 1;
  while (k >= 0) {
    if (s1[i-1] == s2[j-1] && dp[i][j] == dp[i-1][j-1] + 1) {
      rez[k--] = s1[i-1];
      i--, j--;
    } else if (dp[i][j] == dp[i][j-1])
      j--;
    else
      i--;
  }
  return rez;
}
```

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

```
int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1]) {
        dp[j] = prethodni + 1;
      } else {
```

```
        dp[j] = max(dp[j-1], dp[j]);
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}
```

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

Najduža zajednička podniska

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

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

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

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

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

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

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

  return maxPodniska;
}
```

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

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

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

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

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

```
        dp[j] = 0;
      if (dp[j] > maxPodniska)
        maxPodniska = dp[j];
    }

  return maxPodniska;
}
```

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

Edit-rastojanje

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

Izvedimo prvo induktivno-rekurzivnu konstrukciju.

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

Na osnovu ovoga lako možemo definisati rekurzivnu funkciju koja izračunava edit-rastojanje. Da nam se niske ne bi menjale tokom rekurzije (što može biti

sporo), efikasnije je da niske prosleđujemo u neizmenjenom obliku i da samo prosleđujemo brojeve karaktera njihovih prefiksa koji se trenutno razmatraju.

```
int editRastojanje(const string& s1, int n1,
                   const string& s2, int n2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  if (n1 == 0 && n2 == 0)
     return 0;
  if (n1 == 0)
     return n2 * cenaUmetanja;
  if (n2 == 0)
     return n1 * cenaBrisanja;
   if (s1[n1-1] == s2[n2-1])
     return editRastojanje(s1, n1-1, s2, n2-1);
   else {
     int r1 = editRastojanje(s1, n1-1, s2, n2) + cenaBrisanja;
     int r2 = editRastojanje(s1, n1, s2, n2-1) + cenaUmetanja;
     int r3 = editRastojanje(s1, n1-1, s2, n2-1) + cenaIzmene;
     return min({r1, r2, r3});
   }
}

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

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

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

  dp[0][0] = 0;
  for (int i = 0; i <= n1; i++)
    dp[i][0] = i * cenaBrisanja;
```

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

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

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

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

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

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

```
bravo!
```

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

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

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

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

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

čuvati vrednost sa polja $(i-1, j-1)$, a promenljiva `tekuci` će čuvati vrednost sa polja $(i-1, j)$.

```
int editRastojanje(const string& s1, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1);
  for (int j = 0; j <= n2; j++)
    dp[j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    dp[0] = i * cenaBrisanja;
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni;
      else {
        int r1 = tekuci + cenaBrisanja;
        int r2 = dp[j-1] + cenaUmetanja;
        int r3 = prethodni + cenaIzmene;
        dp[j] = min({r1, r2, r3});
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}
```

Najduži palindromski podniz

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

Krenimo od rekurzivnog rešenja.

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

bez poslednjeg karaktera. Duži od ta dva palindromska podniza je traženi palindromski podniz cele niske.

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

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

Na osnovu ovoga, funkciju je veoma jednostavno implementirati.

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

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

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

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

```
}

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

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

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

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

Konačno rešenje 6 odgovara podnizu `abccba`.

```
int najduziPalindrom(const string& s) {
  int n = s.length();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i].resize(n, 0);
    dp[i][i] = 1;
  }
  for (int r = 1; r < n; r++)
    for (int p = 0; p + r < n; p++) {
      int q = p + r;
      if (s[p] == s[q])
        dp[p][q] = dp[p+1][q-1] + 2;
      else
        dp[p][q] = max(dp[p+1][q], dp[p][q-1]);
    }

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

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

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

```
int najduziPalindrom(const string& s) {
  int n = s.length();
  // elementi dve prethodne dijagonale
  vector<int> dpp(n, 0);
  vector<int> dp(n, 1);
  for (int r = 1; r < n; r++) {
    for (int p = 0; p + r < n; p++) {
      int q = p + r;
      if (s[p] == s[q]) {
        int tmp = dp[p];
        dp[p] = dpp[p+1] + 2;
        dpp[p] = tmp;
      }
      else {
        dpp[p] = dp[p];
        dp[p] = max(dp[p], dp[p+1]);
      }
    }
    dpp[n-r] = dp[n-r];
  }

  return dp[0];
}
```

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

Najduži rastući podniz

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

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

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

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

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

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

  vector<int> dp(n);
  dp[0] = 1;
  for (int i = 1; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++)
```

```
      if (a[i] > a[j] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;
  }

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

  return max;
}
```

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

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

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

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

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

```
    // odredjujemo prethodnu poziciju na kojoj se završava
    // niz koji je produžen elementom a[i]
    for (int j = 0; j < i; j++) {
      if (a[i] > a[j] && dp[i] == dp[j] + 1) {
        i = j;
        break;
      }
    }
  }
}
```

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

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

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

Efikasnije rešenje

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

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

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

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

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

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

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

Ključni dobitak nastaje kada se primeti da, pošto su elementi u tabeli sortirani, poziciju prvog elementa koji je veći ili jednak od tekućeg možemo ostvariti bi-

narnom pretragom. Otuda sledi naredna efikasna implementacija (u nizu `dp` vrednost najmanjeg završnog elementa za nizove dužine d pamtimo na poziciji $d-1$).

```
int najduziRastuciPodniz(const vector<int>& a) {
  // broj elemenata niza a
  int n = a.size();
  // za svaku dužinu niza d na poziciji d-1 pamtimo vrednost najmanjeg
  // elementa kojim se može završiti podniz dužine d tekućeg dela niza
  vector<int> dp(n);
  // dužina najdužeg rastućeg podniza tekućeg dela niza
  int max = 0;
  // obrađujemo redom elemente niza
  for (int i = 0; i < n; i++) {
    // binarnom pretragom nalazimo poziciju prvog elementa u nizu dp
    // koji je veći ili jednak od tekućeg
    auto it = lower_bound(dp.begin(), next(dp.begin(), max), a[i]);
    int d = distance(dp.begin(), it);

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

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

  return max;
}
```

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

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

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

koji se nalazi u tabeli pre njega.

```
void najduziRastuciPodniz(const vector<int>& a, vector<int>& podniz) {
  // dužina niza a
  int n = a.size();
  // za svaku dužinu niza d na poziciji d-1 pamtimo vrednost najmanjeg
  // elementa kojim se može završiti podniz dužine d tekućeg dela niza
  vector<int> dp(n);
  // pozicije prethodnih elemenata u rastućim nizovima
  vector<int> prethodni(n, -1);
  // dužina najdužeg rastućeg podniza tekućeg dela niza
  int max = 0;
  // obrađujemo redom elemente niza
  for (int i = 0; i < n; i++) {
    // binarnom pretragom nalazimo poziciju prvog elementa u nizu dp
    // koji je veći ili jednak od tekućeg
    auto it = lower_bound(begin(dp), next(begin(dp), max), i,
                          [a](int x, int y) {
                            return a[x] < a[y];
                          });
    int d = distance(dp.begin(), it);

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

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

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

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

Najveći kvadrat u matrici

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

od jedinica.

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

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

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

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

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

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

Na osnovu prethodne diskusije, rekurzivna implementacija sledi direktno.

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

    if (A[i][j] == 0)
       return 0;
```

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

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

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

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

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

```
int maxKvadrat(int A[MAX][MAX], int m, int n) {
```

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

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

Optimalni raspored zagrada

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

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

zagrada u nekom potproblemu, bolje globalno rešenje možemo dobiti zamenog tog rasporeda optimalnim). Dakle, potprobleme možemo rešavati rekurzivnim pozivima.

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

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

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

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

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

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

```
int minBrojMnozenja(const vector<int>& dimenzije, int l, int d, vector<vector<int>>& memo) {
  if (memo[l][d] != -1)
```

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

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

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

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

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

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

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

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

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

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

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

Najveći pokupljeni zbir

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

Rešenje možemo ostvariti grubom silom, tj. proverom svih mogućih putanja. Naredna funkcija određuje najveći zbir koji se može postići ako se sa polja (v, k)

stiže na polje $(n-1, n-1)$ u matrici M dimenzije n. Ako je početno polje jednako ciljnom, tada se može pokupiti samo broj sa tom polju. Ako polje u poslednjoj vrsti, tada se do poslednjeg polja stiže samo preko polja desno od njega, a ako je u poslednjoj koloni, onda se stiže samo polja ispod njega. U suprotnom se bira da li će se do kraja stići praveći korak desno ili korak dole. U svim tim slučajevima rekurzivno računamo optimum polja na koje smo prešli (birajući bolju od dve mogućnosti, ako nismo na rubu matrice). Naglasimo da je ovde ispunjen uslov optimalne podstrukture, jer kada bismo od polja na koje smo prešli imali neki neoptimalni put, mogli bismo ga zameniti optimalni i time popraviti rešenje za polje sa kog smo krenuli.

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

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

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

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

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

```
}
```

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

Pretpostavimo da je ulazna matrica jednaka

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

Tada je matrica dinamičkog programiranja u drugom rešenju jednaka

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

Najveći mogući zbir koji se može pokupiti je 38.

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

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

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

0-1 problem ranca

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

Krenimo od rekurzivnog rešenja kakvo smo implementirali u sklopu proučavanja pretrage i odsecanja. Funkcija vraća najveću cenu koja se može postići za ranac date nosivosti ako se posmatra samo prvih n predmeta. Rešenje je veoma jednostavno i zasnovano na induktivno-rekurzivnom pristupu. Bazu čini slučaj $n = 0$, kada je maksimalna moguća cena jednaka nuli jer nemamo predmeta koje bismo uzimali. Kada je $n > 0$ izdvajamo poslednji predmet i razmatramo mogućnost da on nije ubačen i da jeste ubačen u ranac. Drugi slučaj je moguć samo ako je masa tog predmeta manja ili jednaka od nosivosti ranca. Veća od te dve cene predstavlja optimalnu cenu. Rekurzivnim pozivima se traži optimum za skup bez poslednjeg predmeta, što je u redu, jer je za globalni optimum neophodno i da su predmeti iz tog podskupa odabrani optimalno (zadovoljen je uslov optimalne podstrukture).

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

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

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

Ovim smo dobili algoritam čija je memorijska složenost $O(M)$ gde je M nosivost ranca, dok je velika složenost $O(N \cdot M)$ gde je N broj predmeta, a M nosivost

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

Optimalno pakovanje

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

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

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

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

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

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

```
      }
      // dodajemo trenutnog u lift
      uLiftu += tezine[permutacija[i]];
    }
    // ažuriramo minimalan broj vožnji
    if (brojVoznji < minBrojVoznji)
      minBrojVoznji = brojVoznji;

    // prelazimo na narednu permutaciju
  } while (next_permutation(begin(permutacija), end(permutacija)));

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

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

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

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

Implementacija može biti sledeća.

```
int brojVoznji(vector<int>& tezine, int nosivost) {
  int n = tezine.size();
  // sortiramo po težini nerastuće
  sort(begin(tezine), end(tezine), greater<int>());
  // beležimo da li se osoba i već odvezla
  vector<bool> odvezaoSe(n, false);
  // broj preostalih osoba koje čekaju lift
  int preostalo = n;
  // trenutna masa ljudi u liftu
  int uLiftu = 0;
  // potreban broj vožnji
  int broj = 1;
  // dok se nisu svi prevezli
  while (preostalo > 0) {
```

```
    // tražimo najtežeg od preostalih koji može da stane u lift
    int i;
    for (i = 0; i < n; i++)
      if (!odvezaoSe[i] && uLiftu + tezine[i] <= nosivost) {
        // osoba i ulazi u lift
        odvezaoSe[i] = true;
        preostalo--;
        uLiftu += tezine[i];
        break;
      }
    // ako niko od preostalih nije mogao da stane u lift
    // lift odlazi i pokrećemo novu vožnju
    if (i == n) {
      broj++;
      uLiftu = 0;
    }
  }
  return broj;
}
```

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

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

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

opterećena). Najmanji broj vožnji za n osoba dobijamo kao minimum potrebnih brojeva vožnji za svaki izbor poslednje osobe. Ostaje još pitanje kako je moguće odrediti najmanji moguću opterećenost poslednje vožnje pri tom optimalnom broju vožnji i ovde treba pažljivo razmisliti.

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

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

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

Izračunate vrednosti su nam sasvim dovoljne i da nađemo globalni optimum. Naime, u globalno optimalnoj vožnji neka se osoba mora voziti

poslednja. Prilikom analize situacije u kojoj ta osoba ulazi u lift poslednja umećemo da izračunamo koliki je minimalni broj vožnji i kolika je minimalna masa u poslednjoj vožnji u kojoj ta osoba učestvuje. Međutim, to će upravo biti minimalna moguća masa u poslednjoj vožnji od svih situacija u kojima se n osoba prevozi u optimalnom broju vožnji.

Pošto se u opisanoj induktivno-rekurzivnoj konstrukciji isti rekurzivni pozivi vrše više puta, efikasnost možemo popraviti dinamičkim programiranjem. Argument svakog rekurzivnog je neki podskup polaznog skupa osoba. Postavlja se pitanje kako da znamo da smo neki takav podskup već ranije obrađivali tj. kako da u programu predstavljamo takve podskupove. Najjednostavniji način je da koristimo kodiranje podskupova pomoću neoznačenih brojeva - bit na poziciji i će biti 1 ako i samo ako se osoba i nalazi u podskupu. Tada se u dinamičkom programiranju za pamćenje rezultata može koristiti običan niz koji je indeksiran kodovima podskupa. Još jedan trik koji malo olakšava implementaciju je da iz rekurzije izađemo još jedan korak kasnije i da za skup od 0 osoba kažemo da je minimalan broj vožnji 1 (ne 0) i da je najmanja masa u poslednjoj vožnji jednaka 0.

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

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

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

```
          // osoba i staje u poslednju vožnju
          // najmanja težina poslednje vožnje u osobe i jednaka
          // je minI.second + tezine[i]
          minI.second += tezine[i];
        } else {
          // osoba i ne staje u poslednju vožnju
          // povećavamo potreban broj vožnji
          minI.first++;
          // minimalna težina kada je osoba i u poslednjoj vožnji
          minI.second = tezine[i];
        }
        // ažuriramo globalni minimum
        ret = min(ret, minI);
      }
    }

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

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

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

```
int brojVoznji(const vector<int>& tezine, int nosivost) {
  // ukupan broj osoba
  int n = tezine.size();
  // rezultati za sve podskupove
  vector<pair<int, int>> dp(1 << n);
  // rezultat za prazan podskup
  // potrebna je jedna vožnja i masa 0 u poslednjoj vožnji
  dp[0] = {1, 0};
  // obrađujemo neprazne podskupove u rastućem redosledu
  // binarnih kodova
  for (unsigned podskup = 1; podskup < (1<<n); podskup++) {
    // minimum inicijalizujemo na "beskonačno"
```

```
    // sigurni smo da će rešenje biti bolje od ovoga
    dp[podskup] = {n, nosivost};
    for (int i = 0; i < n; i++) {
      // preskačemo osobe koje nisu u podskupu
      if (podskup & (1 << i)) {
        // rešenje za podskup bez osobe i
        auto minI = dp[podskup ^ (1 << i)];
        if (p.second + tezine[i] <= nosivost)
          // osoba i staje u poslednju vožnju
          // najmanja težina poslednje vožnje u osobe i jednaka
          // je minI.second + tezine[i]
          minI.second += tezine[i];
        else {
          // osoba i ne staje u poslednju vožnju
          // povećavamo potreban broj vožnji
          minI.first++;
          // minimalna težina kada je osoba i u poslednjoj vožnji
          minI.second = tezine[i];
        }
        dp[podskup] = min(dp[podskup], minI);
      }
    }
  }
  // rezultat za skup u kome je svih n osoba
  return dp[(1<<n)-1].first;
}
```

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

Čas 12.1, 12.2, 12.3 - pohlepni (gramzivi) algoritmi

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 (engl. greedy algorithms). Pohlepni algoritmi ne vrše ispitivanje različitih slučajeva niti iscrpnu pretragu i stoga su po pravilu veoma efikasni (pretraga je isečena na osnovu određenih meta-argumenata). Takođe, obično se veoma jednostavno implementiraju. Sa druge strane, potrebno je dokazati da se pohlepnim algoritmom dobija korektno (optimalno) 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.

Algoritmi zasnovani na pretrazi ili na dinamičkom programiranju obično u svakom koraku razmatraju više mogućnosti (kojima se dobija više potproblema) i nakon razmatranja svih mogućnosti biraju onu najbolju. Dakle, izbor se vrši tek nakon rešavanja potproblema. Za razliku od toga gramzivi algoritmi unapred znaju koja mogućnost će voditi do optimalnog rešenja i izbor vrše odmah, nakon čega rešavaju samo jedan potproblem. U slučaju optmizacionih problema i u slučaju gramzivih algoritma 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 tj. rešenje koje zadovoljava sve uslove zadatka.

Nakon toga je potrebno dokazati i da je rešenje dobijeno strategijom optimalno. Ti dokazi su po pravilu teži i postoji nekoliko tehnika kako se oni izvode. 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 može 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* (engl. exchange).

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* (engl. greedy stays ahead).

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* (engl. structural bound).

Broj žabinih skokova

Problem: Kamenje je postavljeno duž pozitivnog dela x-ose i za svaki kamen je poznata njegova koordinata x. Žaba kreće da skače sa prvog kamena (koji se nalazi u koordinatnom početku) i želi da u što manje skokova dođe do poslednjeg kamena. U svakom skoku ona može da skoči samo u napred i da preskoči najviše

rastojanje r (a može da skoči i manje, ako je to potrebno). Napisati program koji određuje da li žaba može stići do poslednjeg kamena i ako može u koliko najmanje skokova to može učiniti.

Možemo slobodno pretpostaviti da je niz koordinata kamenja sortiran (ako nije, uvek ga možemo na početku sortirati).

Rešenje grubom silom bi podrazumevalo da se u svakom koraku razmotre sve mogućnosti za žabin skok i rekurzivno rešavao problem gde je polazni kamen promenjen. Pošto bi u toj situaciji sigurno došlo do preklapanja rekurzivnih poziva (jer bi žaba na više načina stizala do istog kamena), poželjno bi bilo upotrebiti dinamičko programiranje. Naredni program koristi tehniku memoizacije.

```
// najmanji broj skokova od i-tog kamena
// (-1 ako nije moguće stići do kraja)
int brojSkokova(const vector<int>& kamenje, int i, int r,
                vector<int>& memo) {
  // broj kamenova
  int n = kamenje.size();

  // ako smo stigli do poslednjeg kamena, nije potrebno više skakati
  if (i == n - 1)
    return 0;

  // već smo računali najmanji broj skokova sa ovog kamena
  if (memo[i] != 0)
    return memo[i];

  // najmanji broj skokova - n glumi vrednost +beskonacno
  int min = n;
  // proveravamo sve kamenove na koje žaba može doskočiti
  for (int j = i+1; j < n && kamenje[j] - kamenje[i] <= r; j++) {
    // najmanji broj skokova od kamena j do kraja
    int broj = brojSkokova(kamenje, j, r, memo);
    // ako sa kamena j žaba može stići do kraja u manje skokova od
    // tekućeg minimuma, ažuriramo tekući minimum
    if (broj != -1 && broj + 1 < min)
      min = broj + 1;
  }
  // ako je minimum ostao +beskonačno nije moguće doskočiti do kraja
  if (min == n)
    return memo[i] = -1;
  // pamtimo i vraćamo najmanji broj skokova
  return memo[i] = min;
}

int brojSkokova(const vector<int>& kamenje, int r) {
  // skakanje kreće sa kamena broj 0
  vector<int> memo(kamenje.size(), 0);
```

```
  return brojSkokova(kamenje, 0, r, memo);
}
```

Rešenje se može kreirati i dinamičkim programiranjem naviše. Tada bi se popunjavao niz koji za svaki kamen određuje broj potrebnih skokova i popunjavanje bi teklo od poslednjeg kamena unazad.

```
int brojSkokova(const vector<int>& kamenje, int r) {
  int n = kamenje.size();
  vector<int> dp(kamenje.size());
  dp[n-1] = 0;

  for (int i = n-2; i >= 0; i--) {
    dp[i] = n;
    for (int j = i+1; j < n && kamenje[j] - kamenje[i] <= r; j++)
      if (dp[j] != n && dp[j] + 1 < dp[i])
        dp[i] = dp[j] + 1;
  }

  return dp[0];
}
```

Vremenska složenost pristupa zasnovanog na dinamičkom programiranju je $O(n^2)$.

Međutim, iako u svakom koraku žaba može imati izbor između nekoliko kamenova na koje može da skoči, intuitivno nam je jasno da ona ništa ne gubi time što skoči što dalje. Stoga deluje prilično jasno da žaba u svakom koraku treba da skoči na što dalji kamen koji je na rastojanju manjem od r. Ako takav kamen ne postoji, tada žaba ne može stići do kraja. Nakon što žaba napravi prvi skok, postupak se ponavlja na isti način, sve dok ne stigne do kraja ili dok se ne dođe do situacije u kojoj žaba ne može da skoči dalje (ovo, naravno, ukazuje na induktivno-rekurzivnu prirodu algoritma).

U svakom koraku tražimo najdalji kamen na koji žaba može da skoči i to tako što tražimo prvi kamen na koji žaba ne može da doskoči. Kada takav kamen nađemo (ili kada ustanovimo da žaba može da skoči na svaki kamen ispred sebe, jer smo došli do kraja niza), žaba skače na kamen ispred njega i postupak se nastavlja sve dok žaba ne skoči na poslednji kamen ili dok se ne dogodi da žaba ne može da skoči već na prvi kamen ispred sebe.

```
int brojSkokova(const vector<int>& kamenje, int r) {
   // broj kamenova
   int n = kamenje.size();
   // broj napravljenih skokova
   int broj = 0;
   // redni broj kamena na kome se nalazi žaba
   int kamen = 0;
   // dok žaba ne doskoči na poslednji kamen
```

```
    while (kamen < n - 1) {
        // određujemo redni broj kamena na koji žaba skače
        int noviKamen = kamen;
        while (noviKamen + 1 < n &&
               kamenje[noviKamen + 1] - kamenje[kamen] <= r)
            noviKamen++;
        // žaba ne može da skoči ni na jedan kamen
        if (noviKamen == kamen)
            return n;
        // žaba pravi skok
        broj++;
        kamen = noviKamen;
    }
    // vraćamo izračunati broj skokova
    return broj;
}
```

Iako postoje dve ugnežđene petlje, složenost prethodnog algoritma je $O(n)$ (pokazivači `kamen` i `noviKamen` se nikada ne umanjuju).

Ovaj algoritam je tipičan pohlepni algoritam jer se u svakom koraku uzima što je više moguće i tako da se dolazi i do globalnog optimuma. Ostaje pitanje kako dokazati da je ova strategija korektna.

Prvo dokazujemo da prethodni algoritam uvek daje korektno rešenje tj. rešenje u skladu sa uslovom zadatka (žaba kreće sa prvog kamena, dolazi na poslednji i u svakom koraku skače samo napred, ne više od r metara). Zaista, implementacija je određena tako da je svaki skok koji žaba pravi u prethodnom algoritmu skok na kamen koji je udaljen najviše r (to se u algoritmu eksplicitno proverava), pa smo sigurni da je niz skokova, ako se pronađe, ispravan. Dodatno je potrebno da dokažemo da je i u slučaju kada funkcija vraća -1, taj odgovor korektan tj. da tada zaista nije moguće da žaba dođe do kraja. To se dešava kada postoje dva kamena sa koordinatama x_i i x_{i+1} takvi da je $x_{i+1} - x_i > r$. Ako bi postojalo neko ispravno rešenje, žaba bi morala da skoči sa nekog kamena pre i (ili sa kamena i) na neki kamen nakon x_{i+1} (ili na kamen x_{i+1}), no to je nemoguće jer je zbog sortiranosti niza rastojanje svakog takvog para kamenova strogo veće od r.

Drugo što je potrebno dokazati je da je rešenje koje se dobija strategijom optimalno. U ovom slučaju to znači da pohlepna strategija daje najmanji mogući broj skokova.

U ovom konkretnom slučaju ćemo dokazati da se kamen na kom se žaba nalazi nakon i koraka primene strategije nikada ne nalazi strogo iza kamena na kom se žaba nalazi nakon i koraka u optimalnom rešenju (žaba skače po strategiji je u svakom koraku ili ispred ili na istom kamenu u odnosu na sve žabe koje dostižu cilj u najmanjem broju koraka). Ovo je tipičan dokaz tehnikom *pohlepno rešenje je uvek ispred*. Formalno, neka je optimalna vrednost broja skokova k i i neka je $x_0^*, x_1^*, \ldots, x_{k-1}^*$ sortiran niz x-koordinata kamenja na koje žaba staje

u jednom takvom optimalnom rešenju. Neka je $x_0, x_1, \ldots, x_m$ rešenje dobijeno na osnovu naše strategije. Indukcijom dokazujemo da je $x_i \geq x_i^*$. Bazu čini slučaj $i = 0$ i tada je $x_0 = x_0^*$, jer se žaba u oba slučaja nalazi na početnom kamenu. Pod pretpostavkom da važi $x_i \geq x_i^*$ dokazujemo da važi $x_{i+1} \geq x_{i+1}^*$. Ako je $x_i \geq x_{i+1}^*$, pošto žaba skače samo napred, važi da je $x_{i+1} > x_i \geq x_{i+1}^*$. U suprotnom je $x_i < x_{i+1}^*$. Pošto je optimalno rešenje korektno, znamo da je $x_{i+1}^* - x_i^* \leq r$. Pošto na osnovu induktivne hipoteze znamo da važi $x_i \geq x_i^*$ važi i $x_{i+1}^* - x^i \leq r$. Dakle, žaba sa kamena x_i može sigurno da doskoči na kamen x_{i+1}^* (on se nalazi ispred, na rastojanju manjem od r), a možda može i dalje. Pošto gramziva strategija uzima uvek najdalji skok važi da je $x_{i+1} \geq x_{i+1}^*$. Na osnovu dokazanog važi da je $x_{k-1} \geq x_{k-1}^*$, međutim, pošto je x_{k-1}^* koordinata poslednjeg kamena, to mora biti i x_{k-1}. Dakle, i optimalna strategija stiže do poslednjeg kamena u k koraka, pa optimalno rešenje nije bolje od pohlepnog.

Provera podniza

Problem: Date su dve reči zapisane malim slovima. Napisati program kojim se proverava da li slova druge reči čine podniz slova prve reči, tj. da li se druga reč može dobiti izbacivanjem nekih slova (ne obavezno susednih) iz prve reči. Dokazati korektnost algoritma.

Pri precrtavanju slova, njihov redosled se ne menja što znači da i u drugoj reči redosled slova mora biti isti kao u prvoj. Svako slovo druge reči mora da se javi u prvoj. Ako postoji rešenje problema, onda se rešenje može dobiti i tako što se u prvoj reči zadrži prvo pojavljivanje prvog slova druge reči (sva slova ispred njega se precrtaju) i ostala slova druge reči potraže iza tog prvog pojavljivanja. Naime, ako bi postojalo rešenje u kojem je prvo pojavljivanje prvog slova druge reči u prvoj reči precrtano, a zadržano je neko njegovo kasnije pojavljivanje, pošto se sva ostala zadržana slova druge reči u prvoj reči nalaze iza tog kasnijeg pojavljivanja prvog slova, mogli bismo precrtati to zadržano kasnije pojavljivanje, a zadržati prvo pojavljivanje i tako dobiti ispravno precrtavanje u kojem je zadržano baš prvo pojavljivanje. Sličan je slučaj i sa daljim slovima (uvek u prvoj reči biramo prvo neprecrtano pojavljivanje tekućeg slova druge reči, jer ako reč možemo dobiti precrtavanjem slova, možemo je dobiti i korišćenjem te strategije). Stoga se rešenje zasniva na jednostavnom pohlepnom algoritmu. Primetimo da smo prethodnim razmatranjem zapravo dokazali korektnost ove strategije, tako što smo pokazali da se bilo koje drugo ispravno rešenje može korak po korak transformisati u ono dobijeno pohlepnom strategijom (upotrebili smo dokaz zasnovan na tehnici *razmene*).

Na osnovu ovoga, dovoljno prolaziti u petlji kroz prvu reč, karakter po karakter, i proveravati da li je tekući karakter jednak karakteru koji je na redu u drugoj reči (na početku, na redu je prvi karakter sa indeksom 0). Ako su odgovarajući karakteri jednaki, prelazi se na naredni karakter u obe reči, a inače samo u prvoj. Petlju treba prekinuti kada prođemo kroz sve karaktere bar jedne reči. Ako smo

prošli kroz sve karaktere druge reči, zaključujemo da se druga reč može dobiti precrtavanjem slova prve, inače ne.

```
bool jePodniz(const string& s1, const string& s2) {
  // redom prolazimo kroz slova obe reči dok ne dodjemo do kraja jedne
  // od njih
  int i = 0, j = 0;
  while (i < s1.size() && j < s2.size()) {
    // ako je tekuće slovo prve reci jednako tekućem slovu druge reči
    // onda ga zadržavamo, a u suprotnom ga precrtavamo
    if (s1[i] == s2[j])
      // ako smo slovo zadržali, prelazimo na naredno slovo druge reči
      j++;
    // prelazimo na naredno slovo prve reči
    i++;
  }

  // reč se moze dobiti ako i samo ako smo stigli do kraja druge reči
  // (sva slova druge reči smo pronasli u prvoj)
  return j == s2.size();
}
```

Drugi pogled na isti ovaj postupak je da se za svaki karakter druge reči proveri da li postoji u prvoj reči i ako postoji, pronađe njegovo prvo pojavljivanje, pri čemu se pretraga za prvim karakterom vrši od početka prve reči, a za svakim narednim od pozicije iza one na kojoj je prethodni karakter nađen. Ako se neki karakter ne može pronaći, tada drugu reč nije moguće dobiti od prve. Ako se svi karakteri druge reči uspešno pronađu, onda je drugu reč moguće dobiti od prve. Pretragu karaktera možemo vršiti ili algoritmom linearne pretrage ili bibliotečkom metodom `find` koja vraća specijalnu vrednost `string::npos` ako karakter nije nađen.

```
bool jePodniz(const string& s1, const string& s2) {
  // tražimo jedno po jedno slovo druge reči u prvoj, krenuvši od pozicije i
  int i, j;
  for (i = 0, j = 0; j < s2.size(); i++, j++) {
    // trazimo tekuce slovo druge reči u prvoj, krenuvši od pozicije i
    i = s1.find(s2[j], i);
    // ako ga nismo našli, tada prekidamo pretragu
    if (i == string::npos)
      break;
    // prelazimo na sledeću poziciju u obe reči
  }
  return j == s2.size();
}
```

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 i dokazati njegovu korektnost.

I ovaj problem je moguće rešiti pohlepnom strategijom. 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 (jer je on viši od svih mladića) i dobiti isti broj plesnih parova (primetimo da u ovom razmatranju koristimo tehiku *razmene*). 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 (jer sa njima niko ne može da pleše), 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 (one koje su do sada uparene i one koje nisu uparene, ali su više od tekućeg momka). 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, dakle, dominira složenost sortiranja, pa je ukupna složenost $O(n \log n)$.

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

using namespace std;

int main() {
   // ucitavamo ulazne podatke
   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];

    // želimo da i devojke i mladiće obilazimo u opadajućem redosledu visine
    sort(begin(momci), end(momci), greater<int>());
    sort(begin(devojke), end(devojke), greater<int>());
    int brojParova = 0;
    // tekući indeks momka i devojke
    int m = 0, d = 0;
    while (true) {
        // trazimo najvišu devojku sa kojom može da pleše momak na poziciji m
        while (d < n && momci[m] < devojke[d])
           d++;
        // ako takva devojka ne postoji, ne možemo povećati broj parova
        if (d >= n) break;
        // našli smo par
        brojParova++;
        // prelazimo na narednog momka i devojku
        m++, d++;
    }
    // ispisujemo konačan rezultat
    cout << brojParova << endl;

    return 0;
}
```

Dokažimo i formalno korektnost (koristimo tehniku razmene). Rešenje koje prethodni algoritam daje zadovoljava uslove zadatka jer se svaki momak uparuje sa devojkom koja nije viša od njega (to se eksplicitno proverava) i nakon uparivanja se eliminiše iz razmatranja, tako da smo sigurni da zaista postoji korektno uparivanje za broj parova koji se vraća. Pokažimo i da naša strategija pravi optimalni broj parova. Dokaz će ići tehnikom razmene, tj. time što ćemo se pokazati da se optimalno uparivanje može transformisati u ono dobijeno gramzivom strategijom, održavajući ukupan broj formiranih parova. Posmatrajmo neko optimalno uparivanje. Neka je m_i najviši momak koji učestvuje u uparivanju. Ako on nije ukupno najviši momak m_s, tada najviši momak sigurno nije uparen. Možemo momka m_i izbaciti iz uparivanja i njemu pridruženu devojku d_i pridružiti ukupno najvišem momku m_s (to je moguće jer je $m_s \geq m_i \geq d_i$). Takvo uparivanje je i dalje optimalno (jer se broj parova nije promenio). Neka je d_s devojka koja bi bila odabrana strategijom (najviša devojka koja nije viša od d_s, tj. najviša devojka za koju važi $m_s \geq d_s$). Ako ona nije angažovana u trenutnom uparivanju, onda devojku d_i koja trenutno pleše sa m_s možemo izbaciti i zameniti njome (to je moguće jer je $m_s \geq d_s$). Ako jeste raspoređena da pleše sa nekim m_j, onda možemo napraviti razmenu tako da m_s pleše sa d_s, a m_j sa d_i. Dokažimo da je ovo i dalje korektno uparivanje. Važi da je $m_s \geq d_s$

i $m_s \geq d_i$. Pošto je d_s najviša devojka koja može da igra sa m_s, važi da je $d_s \geq d_i$. Zato je $m_j \geq d_s \geq d_i$. Sa ove dve eventualne razmene dobijamo i dalje optimalan raspored koji je u skladu sa našom strategijom što se tiče prvog plesnog para. Nastavljajući razmene po istom principu (tj. na osnovu induktivnog argumenta), uparivanje možemo transformisati u ono formirano našom strategijom, zadržavajući sve vreme optimalnost.

Dualno rešenje bi bilo ono u kome obrađujemo devojke u rastućem redosledu visine i svakoj devojci dodeljujemo što nižeg momka koji može da pleše sa njom.

Mentori

Problem: Šahisti jednog tima se spremaju za dolazeće turnire. Odlučeno je da šahisti sa boljim rejtingom pomažu onim sa lošijim rejtingom, tako što će im biti mentori. Da bi jedan šahista mogao da bude mentor drugome, njegov rejting treba da bude bar dva puta veći nego rejting prvoga. Ako su poznati rejtinzi svih šahista, napisati program koji određuje koji je najveći broj parova učenik-mentor koji se može formirati, pri čemu šahista može istovremeno biti i učenik i mentor (mentor onome ko ima bar dva puta manji rejting od njega, a učenik onome koji ima bar dva puta veći rejting od njega), ali ni jedan šahista ne može da ima dva učenika niti dva mentora. Dokazati korektnost.

Primetimo sličnost ovog zadatka sa prethodnim (ponovo je u pitanju određivanje maksimalnog uparivanja dva uređena niza, jedino što je sada uslov da se a može upariti sa b umesto $a \geq b$, uslov $a \geq 2b$).

Jedan način da dođemo do rešenja je da primetimo da ako možemo odrediti mentore za nekih k takmičara, onda ti isti mentori mogu biti dodeljeni i najslabijim takmičarima. Dakle, rešenju možemo pristupiti tako što pokušavamo redom da dodelimo mentore najslabijim takmičarima, što podrazumeva da niz rejtinga najpre sortiramo i da niz takmičara obilazimo u rastućem redosledu. Svakom narednom takmičaru pokušavamo da dodelimo mentora i to radimo tako da od svih potencijalnih mentora (takmičara koji imaju bar dvostruko veći rejting od njega koji nisu već ranije dodeljeni nekome kao mentori) pronalazimo onog sa najmanjim rejtingom. Takvim izborom ostavljamo mogućnost da potencijalni mentori sa još većim rejtingom kasnije postanu mentori jačim takmičarima.

Prilikom implementacije potrebno je održavati skup dodeljenih mentora (najjednostavnije tako što održavamo asocijativni niz logičkih vrednosti tako da je na mestu k vrednost tačno ako i samo ako je takmičar sa rednim brojem k već dodeljen nekome kao mentor, čime u konstantnom vremenu za svakog takmičara možemo proveriti da li je slobodan da postane mentor). Jedna važna opaska za efikasnu implementaciju je to da ako je učeniku u dodeljen mentor m, tada mentori učenika $u+1$ mogu biti isključivo takmičari sa rednim brojem $m+1$ pa naviše (naime, ako bi neki takmičar zaključno sa rednim brojem m mogao biti mentor učeniku $u+1$, onda bi on sigurno mogao biti mentor i učeniku u, a pošto svakom učeniku dodeljujemo najslabije mentore, on bi morao biti dodeljen učeniku u,

što je kontradikcija sa time da je na raspolaganju da postane mentor učeniku $u + 1$). Na ovaj način se promenljive koje vrše iteraciju kroz učenike i kroz mentore samo uvećavaju (nema potrebe ni u jednom trenutku ih umanjivati), pa je složenost faze pridruživanja linearna i algoritmom dominira složenost sortiranja koja je $O(n \log n)$. Varijanta u kojoj se ova optimizacija ne koristi (koja za svakog učenika u proverava skup potencijalnih mentora iz početka, krenuvši od učenika $u + 1$) dovodi do kvadratne složenosti faze pridruživanja, što kvari složenost celog zadatka na $O(n^2)$.

Dualno, možemo takmičare obilaziti u opadajućem redosledu rejtinga i svakom od njih pokušavati da dodelimo ulogu mentora i to baš najjačem takmičaru među onima koji su bar dvostruko slabiji od njega i koji do tada nisu proglašeni za nečijeg učenika (argumenti su slični: ako možemo da napravimo k parova učenik-mentor, onda možemo da promenimo mentore u takvoj dodeli i da ih zamenimo za k najjačih takmičara, a kada je neki takmičar fiksiran da postane mentor, najbolje je dodeliti ga najjačem takmičaru, da bi slabijim kandidatima bila ostavljena veća mogućnost da nekom postanu mentori).

Još jedna mogućnost, koja je ispravna, ali manje intuitivna je da takmčare obrađujemo u rastućem redosledu rejtinga i da za svakog od njih proveravamo da li je uopšte moguće da postane mentor, tako što proveravamo sve takmičare čiji je rejting bar duplo manji od rejtinga tekućeg kandidata i među onima koji još nisu nikome dodeljeni za učenike tražimo onog koji ima najmanji rejting (opet, da bi kasnijim kandidatima bila ostavljena veća šansa da postanu nekome mentori).

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

using namespace std;

int main()  {
  // učitavamo rejtinge učenika
  int n;
  cin >> n;
  vector<int> rejting(n);
  for (int i = 0; i < n; i++)
    cin >> rejting[i];

  // sortiramo učenike po rastućem redosledu rejtinga
  sort(begin(rejting), end(rejting));

  // broj do sada formiranih parova učenik-mentor
  int brojParova = 0;
  // za svakog učenika podatak o tome da li već dodeljen nekome kao mentor
  vector<bool> jeMentor(n, false);
  // trenuti potencijalni mentor
  int mentor = 1;
```

```
  // dodeljujemo mentore najslabijim učenicima, sve dok postoji bar
  // jedan potencijalnih mentor
  for (int ucenik = 0; mentor < n; ucenik++)
    // dokle god postoji potencijalni mentor
    while (mentor < n)
      if (!jeMentor[mentor] && rejting[mentor] >= 2 * rejting[ucenik]) {
    // pronašli smo potencijalnog mentora za tekućeg učenika
    brojParova++;
    jeMentor[mentor] = true;
    break;
      } else
    // prelazimo na sledećeg kandidata za mentora
    mentor++;

  // ispisujemo ukupan broj parova
  cout << brojParova << endl;

  return 0;
}
```

Razlomljeni problem ranca

Problem: U jednoj prodavnici se prodaju slatkiši (bombonice, čokoladice, keksići) "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. Kako to da postigne? Dokazati korektnost.

Pošto sretni dobitnik želi da pokupi što veću vrednost slatkiša, jasno je da treba da krene uzimanje onih slatkiša koji su najvredniji tj. čija je cena po gramu najveća. Ako tom vrstom slatkiša može da ispuni ceo ranac, najbolje mu je da to da uradi (na ovom mestu 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 (ovde se može uočiti induktivno-rekurzivna konstrukcija).

```
double razlomljeniRanac(const vector<int>& cena,
                        const vector<int>& kolicina,
                        int nosivostRanca) {
  // broj vrsta slatkiša
  int n = cena.size();
  // vektor u kome čuvamo jedinične cene i količine svih slatkiša
  vector<pair<double, int>> jedinicnaCenaIKolicina(n);
  for (int i = 0; i < n; i++) {
    // jedinična cena slatkiša slatkiša broj i
    double jedinicnaCena = (double)cena[i] / (double)kolicina[i];
```

```
    jedinicnaCenaIKolicina[i] =
      make_pair(jedinicnaCena, kolicina[i]);
  }
  // sortiramo opadajuće na osnovu jedinične cene
  sort(begin(jedinicnaCenaIKolicina), end(jedinicnaCenaIKolicina),
       greater<pair<double, int>>());
  // ukupna vrednost slatkiša koja se može poneti u rancu
  double ukupnaVrednost = 0.0;
  // obrađujemo slatkiše po opadajućoj vrednosti jedinične cene
  for (int i = 0; nosivostRanca > 0 && i < n; i++) {
    // čitamo jediničnu cenu i količinu slatkiša broj i
    double jedinicnaCena = jedinicnaCenaIKolicina[i].first;
    int kolicina = jedinicnaCenaIKolicina[i].second;
    // uzimamo što više, ali smo ograničeni raspoloživom količinom
    // i preostalom nosivišću ranca
    int uzetaKolicina = min(kolicina, nosivostRanca);
    // preostala nosivost ranca
    nosivostRanca -= uzetaKolicina;
    // ažuriramo ukupnu vrednost
    ukupnaVrednost += uzetaKolicina * jedinicnaCena;
  }
  // vraćamo ukupnu vrednost uzetih slatkiša
  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.

Dužni smo još da dokažemo da naša gramziva strategija dovodi do optimalnog rešenja. Koristićemo metod razmene. Pretpostavimo da smo sortirali slatkiše tako da važi

$$\frac{v_0}{w_0} \geq \frac{v_1}{w_1} \geq ... \geq \frac{v_{n-1}}{w_{n-1}}.$$

Svako rešenje je određeno uzetom masom (u gramima) svakog od slatkiša. Neka je rešenje na osnovu strategije određeno nizom masa $(s_0, s_1, ..., s_{n-1})$, pri čemu za svako $0 \leq i < n$ važi $0 \leq s_i \leq w_i$ i $\sum_{i=0}^{n-1} s_i \leq W$, gde je W ukupna nosivost ranca. Pretpostavimo da je optimalno rešenje određeno nizom masa $(o_0, o_1, ..., o_{n-1})$, pri čemu za svako $0 \leq i < n$ važi $o_i \leq w_i$ i $\sum_{i=0}^{n-1} o_i \leq W$. Pošto je o optimalno rešenje, mora da važi da je $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$. Naime, znamo da će kada se slatkiši uzimaju na osnovu strategije ukupna masa uzetih slatkiša bili ili jednaka nosivosti ranca ili ukupnoj raspoloživoj masi svih slatkiša (kada je ona veća ili jednaka od nosivosti ranca). Ako bi se u optimalnom rešenju sadržala manja masa slatkiša od toga, mogli bismo masu nekog slatkiša da povećamo i tako da dobijemo još bolje rešenje, što je u kontradikciji sa pretpostavkom optimalnosti.

Pretpostavimo da strategija daje rešenje s koje je različito od optimalnog rešenja o. Neka je j prva pozicija na kojoj se nizovi s i o razlikuju (za svako $0 \leq i < j$ važi da je $s_i = o_i$). Gramziva strategija uzima redom celokupne raspoložive mase svih predmeta, sve do poslednjeg uzetog predmeta gde se uzima maksimalna masa koja staje u ranac, tako da je jedina mogućnost da je $o_j < s_j$ i to za $j < n-1$ (zbog uslova $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$). Razmotrimo rešenje o' koje dobijamo tako što umesto mase o_j predmeta j uzmemo masu s_j (to sigurno možemo, jer je $s_j \leq w_j$). Time smo povećali ukupnu masu u rancu za $s_j - o_j$ i zato ukupnu masu uzetih predmeta nakon pozicije j moramo da smanjimo za $s_j - o_j$ (to možemo jer je $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$ i $\sum_{i=0}^{j-1} o_i = \sum_{i=0}^{j-1} s_i$). Dokažimo da ovo rešenje ne može biti lošije od optimalnog. Njegova vrednost se povećala za $(s_j - o_j) \cdot \frac{v_j}{w_j}$ i umanjila za $\sum_{k=j+1}^{n-1} c_k \frac{v_k}{w_k}$, gde je $\sum_{k=j+1}^{n-1} c_k = s_j - o_j$. Dokažimo da je povećanje veće nego smanjenje. Važi da je

$$\sum_{k=j+1}^{n-1} c_k \frac{v_k}{w_k} \leq \sum_{k=j+1}^{n-1} c_k \frac{v_{j+1}}{w_{j+1}} = \frac{v_{j+1}}{w_{j+1}} \sum_{k=j+1}^{n-1} c_k = \frac{v_{j+1}}{w_{j+1}}(s_j - o_j) \leq \frac{v_j}{w_j}(s_j - o_j).$$

Ovom promenom smo, dakle, napravili rešenje o' čija je vrednost jednaka optimalnoj (jer nijedno rešenje ne može biti bolje od optimalnog rešenja o, a upravo smo dokazali da rešenje o' nije lošije od njega), a koje se poklapa sa strategijskim rešenjem s na jednoj poziciji više nego polazno optimalno rešenje o. Nastavljajući postupak na isti način dobićemo rešenje koje je optimalno i jednako je s.

Vraćanje kusura

Problem: U Srbiji se koriste apoeni od 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000 i 5000 dinara. Napisati program koji formira datih iznos dinara od što manjeg broja novčanica i novčića. Šta bi se desilo da postoji i novčić od 4 dinara?

Jedan direktan način da se problem reši je da se ispitaju sva moguća razlaganja datog iznosa na zbirove sastavljene od ovih brojeva i da se pronađe onaj zbir koji ima najmanji broj novčića (jednostavnosti radi, zanemarimo razliku između novčića i novčanica). Rešenje ovog tipa bi se moglo zasnovati na dinamičkom programiranju kombinovanom sa odsecanjem tokom pretrage. Takvo rešenje je opšte i već smo ga implementirali u poglavlju o dinamičkom programiranju, kada ništa nismo znali o konkretnim apoenima. Dokaz korektnosti ne bi bio težak, ali bi ovaj pristup bio prilično komplikovan i neefikasan. Naime, i bez formalnog matematičkog objašnjenja, svaki prodavac u prodavnici i na pijaci zna da se optimalno rešenje dobija tako što se u svakom trenutku vraća najveći apoen koji je manji ili jednak od trenutnog iznosa i nakon toga se isti princip primenjuje

na preostali iznos sve dok se ne vrati ceo kusur (u pitanju je, dakle, induktivno-rekurzivna konstrukcija). Ovo rešenje je veoma efikasno, lako se implementira, međutim, dokaz njegove korektnosti nije nimalo očigledan. Naime, postojanje novčića od 4 dinara bi pokvarilo situaciju. 8 dinara bi se moglo dobiti od dva novčića od 4 dinara, dok bi gramziva strategija upotrebila tri novčića (od 5, 2 i 1 dinar). Dakle, dokaz korektnosti mora da uključi analizu konkretnih apoena koji su u opticaju i male promene ovih apoena mogu da utiču na to da opisani pristup daje ili ne daje uvek optimalno rešenje.

Dokažimo sada korektnost. Jednostavnosti radi, pretpostavićemo da su u opticaju samo apoeni od 1, 2, 5, 10, 20 i 50 dinara (za veće novčiće dokaz ide po istom principu). Metodom razmene dokazaćemo gornje granice broja novčića od svih apoena u optimalnom rešenju.

- Optimalno rešenje ne može da sadrži više od jednog novčića od 1 dinar. Kada bi postojala makar dva novčića od 1 dinar, oni bi mogli biti zamenjeni jednim novčićem od 2 dinara, čime bi se broj upotrebljenih novčića smanjio, što je u kontradikciji sa pretpostavkom da je polazno rešenje optimalno. Potpuno analogno se dokazuje da optimalno rešenje ne može sadržati ni više od jednog novčića od 10 dinara.
- Dalje, optimalno rešenje ne može sadržati više od dva novčića od 2 dinara. Naime, ako bi sadržalo bar tri novčića od 2 dinara, oni bi mogli biti zamenjeni jednim novčićem od 1 i jednim novčićem od 5 dinara, čime bi se dobilo manje rešenje, što je u kontradikciji sa pretpostavkom da je polazno rešenje optimalno. Potpuno analogno, optimalno rešenje ne može da sadrži ni više od dva novčića od 20 dinara.
- Na kraju, u optimalnom rešenju ne može biti više od jednog novčića od 5 dinara. Naime, ako bi postojala bar dva, ona bi mogla biti zamenjena jednim novčićem od 10 dinara čime bi se dobilo manje rešenje, što je kontradikcija.
- U rešenju nije moguće ni da istovremeno postoje dva novčića od 2 dinara i novčić od 1 dinara, jer bi se svi oni mogli zameniti sa jednim novčićem od 5 dinara, što je opet kontradikcija. Analogno važi i za novčiće od 10 i 20 dinara.

Uzevši u obzir prethodna ograničenja, razmotrimo maksimalne iznose sa optimalnim brojem novčića, koji se mogu dobiti korišćenjem samo određenih skupova novčića. Ispostaviće se da su maksimalni iznosi uvek za jedan manji od prvog većeg apoena.

- Novčići od 1 dinar mogu da naprave najviše iznos od 1 dinara (jer se smeju pojaviti samo jednom).
- Novčići od 1 i 2 dinara mogu da naprave najviše iznos od 4 dinara (jer može biti najviše dva novčića od 2 dinara i u tom slučaju se ne sme koristiti i novčić od 1 dinar).
- Novčići od 1, 2 i 5 dinara mogu da naprave najviše iznos od 9 dinara (jer ne može biti više od jednog novčića od 5 dinara, dva novčića od 2 i jednog novčića od 1 dinara, a ako ima dva novčića od 2 dinara, ne sme se javiti i

novčić od 1 dinara).
- Slično, novčići od 1, 2, 5 i 10 dinara mogu da naprave najviše iznos od 19 dinara (jer se 10 dinara može javiti samo jednom, a od 1, 2 i 5 se može napraviti najviše 9).
- Novčići od 1, 2, 5, 10 i 20 mogu da naprave najviše iznos od 49 dinara (jer 1, 2 i 5 mogu da naprave najviše 9, kako smo objasnili, a pošto se 10 dinara javlja najviše jednom, a 20 dinara najviše dva puta, ali ne sva tri takva zajedno, od 10 i 20 se može napraviti najviše 40).

Dokažimo sada da se za svaki pozitivan iznos u optimalnom rešenju mora nalaziti najveći novčić koji je manji ili jednak od tog iznosa (sve navedene konstatacije se odnose samo na optimalna rešenja).

- Za iznos 1 javlja se samo novčić 1.
- Iznosi između 2 i 4 dinara moraju da sadrže novčić 2. Naime, ne može da se javi novčić od 5 dinara, samo od novčića od 1 dinar može da se napravi najviše 1 dinara, pa za iznose od 2 do 4 dinara mora da se javi bar jedan novčić od 2 dinara.
- Iznosi između 5 i 9 dinara moraju da sadrže novčić od 5 dinara. Naime, ne mogu da sadrže novčić od 10 dinara, a pošto se od novčića od samo 1 i 2 dinara može napraviti najviše 4 dinara, za iznose od 5 do 9 dinara mora da se upotrebni novčić od 5 dinara.
- Iznosi između 10 i 19 dinara moraju da sadrže novčić 10. Naime, 20 dinara ne može da se javi, a pomoću novčića od 1, 2 i 5 dinara najviše se može napraviti 9 dinara.
- Iznosi između 20 i 49 dinara moraju da sadrže bar jedan novčić od 20 dinara. Naime, ne mogu da sadrže 50, a samo sa 1, 2, 5 i 10 se može dobiti najviše 19.
- Iznosi preko 50 dinara moraju da sadrže novčić od 50 dinara. Naime, samo sa novčićima od 1, 2, 5, 10 i 20 je moguće napraviti samo 49 dinara.

Dakle, uspeli smo da za svaki iznos pronađemo novčić koji optimalno rešenje mora da sadrži, čime onda uspevamo da smanjimo dimenziju problema i da do rešenja dođemo direktno, bez bilo kakve pretrage i isprobavanja raznih mogućnosti.

```
void stampajKusur(int iznos) {
  vector<int> apoeni {500, 200, 100, 50, 20, 10, 5, 2, 1};
  while (iznos > 0) {
    for (int apoen : apoeni)
      if (iznos >= apoen) {
        cout << apoen << endl;
        iznos -= apoen;
        break;
      }
  }
}
```

Prethodni algoritam je takav da se u svako pojedinačnom koraku uzima tačno

određeni element rešenja i nema potrebe proveravati različite kombinacije da bi se pronašlo optimalno rešenje, što je, naravno, odlika pohlepnih algoritama.

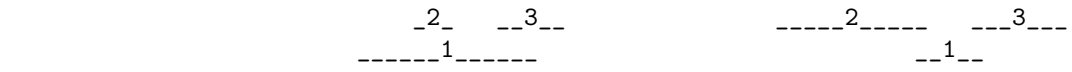
Raspored aktivnosti

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 i dokazati njegovu korektnost.

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.

Razmislimo koji čas ima smisla prvi rasporediti. Iako neko može pomisliti da to može da bude čas koji prvi počinje ili čas koji najkraće traje, ni jedan, ni drugi pristup ne daju optimalno rešenje (kako je ilustrovano sa dva kontra-primera na slici).



U oba primera rasporedio bi se samo čas broj 1, a bolje rešenje je da se umesto njega rasporede časovi 2 i 3.

Pokazaće se da će se optimalno rešenje dobiti 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.

Implementacija je prilično jednostavna.

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

using namespace std;
```

```
// časove predstavljamo uređenim parovima (početak, kraj)
typedef pair<int, int> cas;

// kreiramo čas na osnovu sata i minuta početka i kraja
cas napraviCas(int pocSat, int pocMin, int krajSat, int krajMin) {
  return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}

// očitavamo početak časa
inline int pocetakCasa(const cas& c) {
  return c.first;
}

// očitavamo kraj časa
inline int krajCasa(const cas& c) {
  return c.second;
}

int main() {
  // učitavamo podatke o svim časovima
  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);
  }
  // sortiramo časove po vremenu završetka
  sort(begin(casovi), end(casovi),
       [](const cas& a, const cas& b) {
          return krajCasa(a) < krajCasa(b);
       });

  // prvi čas sigurno može biti održan
  int brojOdrzanihCasova = 1;
  // kraj poslednje održanog časa
  int kraj = krajCasa(casovi[0]);
  // obrađujemo sve časove po redosledu završetka
  for (int i = 1; i < n; i++)
    // ako je čas broj i kompatibilan sa svim ranijim
    if (pocetakCasa(casovi[i]) >= kraj) {
      // njega zakazujemo i ažuriramo broj održanih časova i
      // kraj poslednjeg održanog časa
      brojOdrzanihCasova++;
      kraj = krajCasa(casovi[i]);
    }

  // ispisujemo konačan rezultat
```

```
  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 (ako sa $[s_i, f_i)$ obeležimo interval časa c_i, tada je $f_1 \leq f_2 \leq ... \leq f_k$). Pošto se svi ti časovi mogu održati, između njih nema preklapanja i svaki naredni počinje nakon završetka prethodnog (važi da je $s_{i+1} \geq f_i$). 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 (on već sadrži k elemenata i potrebno je samo pokazati da je taj raspored takođe ispravan, tj. da su svi časovi međusobno kompatibilni). 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 (časovima $c_1, c_2, ..., c_{i-1}$). Potrebno je da pokažemo da je kompatibilan i sa svim časovima koji se održavaju kasnije (časovima $c_{i+1}, ..., c_k$). 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, tj. znamo da je $s_i \geq f_{i-1}$. Zato znamo da se c_i' mora završiti pre c_i tj. da je $f_i' \leq f_i$. Dakle, ako postoje časovi u O pre časa c_i, oni ostaju nepromenjeni i čas c_i' se ne preklapa sa njima (jer je $s_i' \geq f_{i-1}$). Pošto se c_i' ne završava kasnije nego c_i tj. pošto je $f_i' \leq f_i$, on se sigurno ne preklapa ni sa jednim časom iz O koji ide posle c_i (jer svi oni počinju nakon kraja časa c_i tj. nakon trenutka f_i, pa zato počinju i nakon f_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.

Raspored sa najmanjim brojem učionica

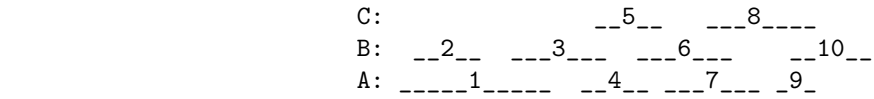
Problem: Za svaki od n časova poznato je vreme početka i završetka. Napiši program koji određuje minimalni broj učionica potreban da se svi časovi održe.

Iako donekle deluje slično prethodnom, rešenje ovog zadatka zahteva drugačiju strategiju. 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.

Naredni primer pokazuje da obilazak u redosledu prvog vremena završetka ne daje uvek optimalno rešenje. Naime, raspored se može napraviti u dve učionice (u učionici A časovi 1 i 4, a u učionici B časovi 2 i 3), a ako bismo učionice obilazili po vremenu završetka, za raspored bi bilo potrebno tri učionice (kako je prikazano na slici).

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. Za prethodni primer dobio bi se raspored prikazan na narednoj slici.

Pogledajmo još jedan primer.



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 (tih d već raspoređenih i čas $[s_j, f_j)$) 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.

Primetimo da je ovaj dokaz drugačiji od ranije viđenih dokaza zasnovanih na tehnici razmene ili na tehnici u kojoj se pokazuje da je pohlepno rešenje uvek ispred po nekom kriterijumu. U ovom dokazu je dokazana granica kvaliteta rešenja (raspored ne može biti u manje učionica nego što je broj časova u najvećoj grupi časova koji se održavaju u nekom trenutku) i zatim je pokazano da pohlepno rešenje dostiže tu granicu.

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 (najlakše tako što tu učionicu izbacimo iz reda i ponovo je dodamo sa ažuriranim vremenom). U suprotnom u red dodajemo novu učionicu kojoj je vreme završetka časa postavljeno na vreme završetka tekućeg časa (broj učionice je za jedan veći od dotadašnjeg broja učionica u redu).

Ukupna složenost algoritma je $O(n \log n)$ - i u fazi sortiranja i u fazi raspoređivanja. Kada bismo umesto reda sa prioritetom koristili običan niz i u njemu stalno tražili minimum, složenost najgoreg slučaja bi porasla na $O(n^2)$.

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

using namespace std;

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);
  }

  // sortiramo časove na osnovu vremena njihovog početka
  sort(begin(casovi), end(casovi),
       [](const Cas& c1, const Cas& c2) {
          return c1.pocetak < c2.pocetak;
       });

  // raspored učionica - svakom času dodeljujemo učionicu
  vector<int> ucionica(n);
  // red u kome čuvamo sve do sada dodeljene učionice, sortirano po redosledu
  // završetka časova u njima
  priority_queue<Ucionica, vector<Ucionica>, PorediUcionice> redUcionica;
  // obilazimo sve časove u redosledu njihovog početka
  for (const Cas& c : casovi) {
    // broj učionice u kojoj će se čas održati
    int brojUcionice;
    if (redUcionica.empty() || redUcionica.top().slobodnaOd > c.pocetak)
      // ako nema slobodnih učionica, otvaramo novu
      brojUcionice = redUcionica.size() + 1;
    else {
      // u suprotnom čas raspoređujemo u onu učionicu koja se najranije ispraznila
      brojUcionice = redUcionica.top().broj;
      redUcionica.pop();
    }
    // beležimo broj učionice za tekući čas
    ucionica[c.broj] = brojUcionice;
    // dodajemo učionicu u red
    redUcionica.push(napraviUcionicu(c.kraj, brojUcionice));
  }

  // ispisujemo ukupni broj učionica i učionice pridružene časovima
  cout << redUcionica.size() << endl;
  for (int u : ucionica)
    cout << u << endl;

  return 0;
```

```
}
```

Raspored sa najmanjim zakašnjenjenjem

Problem: Za svaki od n poslova koje treba završiti poznato je trajanje u minutima i rok do kojeg se posao mora završiti. Svaki posao može da počne u minutu 0. Kvalitet obavljenog posla se ceni na osnovu najvećeg napravljenog zakašnjenja - potrebno je da ono bude što manje. Napisati program koji određuje optimalan raspored i dokazati njegovu korektnost. Na primer, neka su u tabeli data trajanja i rokovi završetka za svaki posao.

```
br  1  2  3  4  5  6
--------------------
ti  1  2  2  3  3  4
di  9  8 15  6 14  9
```

Jedan mogući raspored (napravljen tako da se prvo rade što kraći poslovi) je sledeći:

```
 1 2 2 3 3 4 4 4 5 5 5 6 6 6 6    - broj posla
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5   - minut
```

U njemu se posao 4 završava u minutu 8 i kasni 2 minuta u odnosu na svoj rok 6, dok se posao 6 završava u minutu 15 i kasni 6 minuta u odnosu na svoj rok 9. Najveće kašnjenje u ovom rasporedu je 6 minuta.

Jedan optimalni raspored je sledeći. U njemu se posao 6 završava u minutu 10 i kasni jedan minut u odnosu na svoj rok 9 (to je jedino kašnjenje).

```
 4 4 4 2 2 1 6 6 6 6 5 5 5 3 3   - broj posla
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5  - minut
```

Zadatak je moguće rešiti pomoću gramzivog algoritma. Ključno pitanje je u kom redosledu treba zakazivati poslove. Optimalno rešenje se dobija ako se poslovi zakazuju u redosledu roka njihovog završetka, pri čemu svaki posao počinje odmah nakon što se prethodni završio (ne pravi se pauza ni jedan jedini minut).

Dokažimo da je ova strategija optimalna. Ako su dva posla i i j takvi da je rok završetka prvog r_i pre roka završetka drugog r_j (tj. važi $r_i < r_j$), a da je početak prvog posla s_i zakazan nakon početka drugog posla s_j (tj. važi $s_i > s_j$), reći ćemo da su oni raspoređeni pogrešno.

Pretpostavimo da je O neki optimalni raspored. Ako u tom rasporedu postoje dva pogrešno raspoređena posla, tada u rasporedu sigurno postoje i dva pogrešno raspoređena uzastopna posla. Naime, pretpostavimo da su poslovi i i j raspoređeni naopako. Pošto je rok završetka posla j ispred roka završetka posla i, prilikom obilaska poslova između i i j u redosledu njihovog zakazanog početka, nije moguće da rok njihovog završetka stalno raste i neophodno je

da se prilikom prelaska sa jednog posla na sledeći rok završetka smanji. Dva uzastopna posla kod kojih se to desi su sigurno pogrešno raspoređena.

Dva susedna posla se mogu razmeniti tako da se ostali poslovi ne diraju. Dokažimo da se razmenom dva naopako raspoređena posla maksimalno kašnjenje ne može povećati. Pošto se ostali poslovi ne menjaju razmenom dva susedna naopako raspoređena posla, jedino je relevanto posmatrati njihova kašnjenja. Neka posao i počinje u u trenutku s_i, traje t_i minuta i neka mu je rok završetka d_i. Tada posao j počinje u trenutku $s_i + t_i$, neka traje t_j minuta i neka mu je rok završetka d_j. Kašnjenje posla i je $\max(s_i + t_i - d_i, 0)$ dok je kašnjenje posla j jednako $\max(s_i + t_i + t_j - d_j, 0)$. Pošto su oni raspoređeni naopako važi da je $d_i > d_j$ (i da je $s_i < s_i + t_i$). Nakon razmene njihova kašnjenja postaju $\max(s_i + t_j + t_i - d_i, 0)$ i $\max(s_i + t_j - d_j, 0)$. Posao j je pomeren unapred (za t_i minuta) i njegovo kašnjenje se time samo moglo smanjiti (jer je $\max(s_i + t_j - d_j, 0) \leq \max(s_i + t_i + t_j - d_j, 0)$). Posao i je pomeren unazad, međutim, pošto je $d_i > d_j$, važi da je $\max(s_i + t_i + t_j - d_i, 0) \leq \max(s_i + t_i + t_j - d_j, 0)$, pa posao i ne kasni više nego što je ranije kasnio posao j.

Prethodno opisanom razmenom se broj naopako raspoređenih poslova smanjuje za 1. Razmene možemo ponavljati sve dok ne stignemo do rasporeda u kome nema naopako raspoređenih poslova. Pošto se lako dokazuje da svi rasporedi u kojima nema naopako raspoređenih poslova i nema pauza imaju isto maksimalno kašnjenje (razmenom uzastopnih poslova sa istim rokom ne menja se maksimalno kašnjenje), a naš pohlepni algoritam gradi jedan baš takav raspored, to znači da će se maksimalno kašnjenje našeg rasporeda poklapati sa optimalnim i naš raspored će takođe biti optimalan.

Napomenimo i da naizgled jednostavna modifikacija teksta zadatka dovodi do problema koji nema jednstavno polinomijalno rešenje. Naime, ako bismo umesto najvećeg kašnjenja pokušavali da smanjimo zbir svih kašnjenja dobio bi se NP kompletan problem.

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 (što je upravo slučaj sa standardnim kodovima, kao što je ASCII), 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 $(0 \leq i < n)$ 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 (takvi se kodovi nazivaju prefiksni kodovi). 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 (zato nije moguće da je kôd bilo kog karaktera prefiks koda nekog drugog karaktera).

Za dati skup karaktera i frekvencije njihovog pojavljivanja konstruisati optimalni binarni prefiksni kôd (onaj kod kojeg je broj bitova $\sum_{i=0}^{n-1} f_i b_i$ potreban za zapis celog teksta najmanji). Dokazati optimalnost.

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. U prethodnom primeru, drvo možemo transformisati tako što unutrašnji čvor `01` zamenimo svojim detetom i tako karakteru `b` dodelimo kod `01` umesto `010`. Slično, čvor `11`, a zatim i čvor `1` možemo zameniti svojom decom i tako karakteru `c` možemo dodeliti kôd `1`.

Intuicija nam govori da je poželjno karakterima koji se pojavljuju često dodeljivati kraće kodove. Zato će karakteri koji se javljaju ređe imati duže kodove. Ovo nije teško formalno dokazati. Pretpostavimo da karakter c_i ima frekvenciju pojavljivanja f_i i da se kodira sa b_i bitova, a da karakter c_j ima frekvenciju pojaljivanja f_j i da se kodira sa b_j bitova, pri čemu je $f_i \geq f_j$, dok je $b_i \leq b_j$. Ako se zamene kodovi ta dva karaktera, ukupan potreban broj bitova za kodiranje teksta se ne može povećati. Pošto se ostalim karakterima kodovi ne menjaju, broj bitova zavisi samo od ova dva karaktera. U prvom slučaju on je jednak $f_i b_i + f_j b_j$, a u drugom je jednak $f_i b_j + f_j b_i$. Ako posmatramo razliku $(f_i b_j + f_j b_i) - (f_i b_i + f_j b_j)$ dobijamo izraz $f_i(b_j - b_i) + f_j(b_i - b_j)$ tj. $(f_i - f_j)(b_j - b_i)$. Pošto su oba činioca nenegativna i polazna razlika je nenegativna, pa se ukupan potreban broj bitova ovom transformacijom nije mogao povećati (ako je razlika pozitivna, onda se broj bitova smanjio).

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. Zaista, ako su to karakteri c_1 i c_2 i ako se na mestu najudaljenijem od korena nalaze karakteri c_1' i c_2', tada možemo zameniti c_1 i c_1' i zameniti c_2 i c_2'. Struktura drveta ostaje identična, pa se nakon razmene i dalje dobija ispravan binarni prefiksni kôd. Pošto su nakon ove razmene kodovi zamenjeni tako da su češći karakteri dobili kraće kodove, na osnovu prethodno dokazanog tvrđenja ukupan broj bitova za kodiranje teksta se nije mogao povećati.

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 $f_i + f_j$ (gde su f_i i f_j frekvencije karaktera c_i i c_j). Zaista, ako drvo B ne bi bilo optimalno za modifikovanu azbuku, postojalo bi bolje drvo za nju (recimo B'). U tom drvetu B' list c možemo zameniti unutrašnjim čvorom čija su deca c_i i c_j i tako dobiti drvo A' za polaznu azbuku. Dokažimo da ako bi drvo B' bilo bolje od drveta B, tada bi drvo A' bilo bolje od drveta A, što je kontradikcija. Ukupan broj bitova u drvetu A i B se razlikuje samo za bitove kojima se kodiraju karakteri c_i, c_j i novi karakter c. U drvetu A to je $f_i b + f_j b$ bitova, gde je b broj bitova potreban za kodiranje karaktera c_i i c_j, dok je u drvetu B to $(f_i + f_j)(b-1)$, jer je karakteru c pridružena frekvencija $f_i + f_j$, dok je visina čvora c za 1 manja od visine čvorova c_i i c_j. Dakle, razlika između broja bitova u ta dva drveta je $f_i + f_j$, što je konstanta. Slično, broj bitova za koji se razlikuju A' i B' je takođe $f_i + f_j$. Dakle, ako je broj bitova za B' manji od broja bitova za B, tada bi broj bitova za A' morao da bude manji od broja bitova za A, što je kontradikcija.

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 tj. izlaz iz rekurzije 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).

Potrebno je precizirati još nekoliko detalja da bismo napravili implementaciju. U svakom koraku je u skupu karaktera potrebno određivati dva sa najmanjim frekvencijama i menjati ih sa novim, čija je frekvencija jednaka zbiru frekvencija. Ove operacije se mogu veoma efikasno izvršavati ako se karakteri ubace u red sa prioritetom (hip) sortiran rastuće na osnovu frekvencija. Drugo pitanje je to kako reprezentovati drvo koje se gradi. Unapred znamo da će ono imati $2n-1$ čvorova i znamo da će n karaktera biti listovi tog drveta. Svaki čvor možemo numerisati brojevima od 0 do $2n-2$. Za svaki od $n-1$ unutrašnjih čvorova treba da znamo indeks levog i desnog deteta (uvek postoje oba). Te informacije možemo čuvati u dva pomoćna niza. Unutrašnji čvorovi imaju indekse od n do $2n-2$ i za čvor broj k u prvom nizu na poziciji $k-n$ čuvamo indeks levog deteta, a u drugom nizu na poziciji $k-n$ čuvamo indeks desnog deteta.

Kada je drvo kreirano, kodove svih karaktera možemo dobiti iscrpnim obilaskom celom drveta (rekurzivnom funkcijom).

```
// mapu kodovi popunjava kodovima svih karaktera koji su naslednici čvora i
// čiji je kod jednak niski kod
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)
    // stigli smo do lista
    kodovi[karakteri[i]] = kod;
  else {
    // u pitanju je unutrašnji čvor
    procitajKodove(levo, desno, karakteri, levo[i-n], n, kod + "0", kodovi);
    procitajKodove(levo, desno, karakteri, desno[i-n], n, kod + "1", kodovi);
  }
}

// mapu kodovi popunjava kodovima svih karaktera iz datog drveta
void procitajKodove(const vector<int>& levo,
                    const vector<int>& desno,
                    const vector<char>& karakteri, int n,
                    map<char, string>& kodovi) {
  // čitanje kreće od korena, čiji je kod prazna niska
  procitajKodove(levo, desno, karakteri, 2*n-2, n, "", kodovi);
}

// određuje najmanji broj bitova potrebnih
int brojBitova(const vector<char>& karakteri, const vector<int>& frekvencije) {
  int n = karakteri.size();
  // indeksi dece unutrašnjih čvorova drveta
  vector<int> levo(n-1), desno(n-1);
  // red sa prioritetom u kome čuvamo parove frekvencija i indeksa čvorova, uređen
  // rastuće po frekvencijama
  priority_queue<pair<int, int>, vector<pair<int, int>>,
                 greater<pair<int, int>>> pq;
  // sve karaktere ubacujemo u red
  for (int i = 0; i < n; i++)
    pq.push(make_pair(frekvencije[i], i));

  // određujemo jedan po jedan unutrašnji čvor, sve dok ne završimo sa korenom (2n-2)
  for (int i = n; i < 2*n - 1; i++) {
    // uklanjamo dva karaktera sa najmanjim frekvencijama
    auto f1 = pq.top(); pq.pop();
    auto f2 = pq.top(); pq.pop();
    // dodeljujemo tim karakterima novi unutrašnji čvor
    levo[i - n] = f1.second; desno[i - n] = f2.second;
    // kreiramo novi kombinovani karakter
    pq.push(make_pair(f1.first + f2.first, i));
  }

  // čitamo iz drveta kodove svih karaktera
  map<char, string> kodovi;
  procitajKodove(levo, desno, karakteri, n, kodovi);
```

```
  // svakom karakteru pridružujemo njegovu frekvenciju
  map<char, int> frekvencijeKaraktera;
  for (int i = 0; i < n; i++)
     frekvencijeKaraktera[karakteri[i]] = frekvencije[i];

  // izračunavamo i vraćamo ukupan broj bitova
  int brojBitova = 0;
  for (auto it : kodovi)
    brojBitova += frekvencijeKaraktera[it.first] * it.second.length();
  return brojBitova;
}
```

Vagoni

Problem: Na pruzi se nalaze vagoni označeni brojevima od 1 do n, ali su ispremetani. Potrebno je prebaciti ih na drugi kraj pruge u tako da su poređani od 1 do n. Sa strane pruge nalaze se tri pomoćna koloseka na koji se privremeno mogu smeštati vagoni. Pruga sa pomoćnim kolosecima ima oblik ćiriličkog slova Š sa produženom donjom osnovom.

```
     3 pomoćna koloseka
         |   |   |
ulaz ----+---+---+----  izlaz
```

Vagon se može prebaciti sa ulaza na pomoćni kolosek i sa pomoćnog koloseka na izlaz - nije moguće prebacivanje sa jednog na drugi pomoćni kolosek. Svaki pomoćni kolosek može čuvati proizvoljan broj vagona. Pomoćni koloseci su slepi i redosled izlaska vagona sa pomoćnih koloseka je obratan od redosleda njihovog ulaska.

Razmislimo kako možemo modelovati zadati problem. Vagoni koje treba premestiti se mogu smestiti u red, dok se pomoćni koloseci ponašaju kao stekovi. Čim vagon sa ulaza ili sa nekog pomoćnog koloseka može da se prebaci na izlaz to bez odlaganja treba uraditi (to pomeranje ne može zasmetati ni jednom narednom pomeranju, a odlaganje potencijalno može dovesti do toga da vagon koje je sada moguće prevesti na izlaz postane blokiran). Ključni uvid za rešenje zadatka je da se vagon koji ne može odmah da izađe na izlaz stavlja na onaj pomoćni kolosek na na čijem vrhu se nalazi vagon većom oznakom od oznake tekućeg vagona. U slučaju da ima više takvih bira se onaj sa najmanjom oznakom vagona na vrhu. U slučaju da takvih nema, a postoji prazan kolosek, vagon se postavlja na prazan kolosek. U suprotnom znamo da raspoređivanje nije moguće. Ovo bi trebalo da je intuitivno jasno. Vagon stavljamo na pomoćni kolosek samo kada ne može da pređe direktno na izlaz i cilj je da postavljanje uradimo tako da on u budućnosti što manje smeta budućim vagonima. Ako bi se postavio iznad vagona sa manjom oznakom, došli bismo u situaciju da taj donji vagon nikada ne možemo izvesti na izlaz (jer da bi se vagon sa većom oznakom koji postavljamo iznad njega mogao izvesti, potrebno je da budu izve-

deni svi vagoni sa manjim oznakama od njega, pa i ovaj vagon koji je njime upravo zaklonjen). Dakle, vagon moramo postaviti ili na onaj kolosek na čijem vrhu je vagon sa većim brojem ili na prazan kolosek. Taj vagon će sigurno ograničiti jedan pomoćni kolosek tako da na njega mogu da stanu samo brojevi sa manjim oznakama. Potrebno je jedino voditi računa da druga dva koloseka budu što manje ograničena tj. vagon treba postaviti na onaj kolosek koji je već sada najviše ograničen, jer se tako u budućnosti ništa neće pokvariti. Naime, svaki vagon koji bi mogao da bude postavljen na taj najviše ograničen kolosek, moći će da bude postavljen i na druge koloseke (jer su oni manje ograničeni). Možemo smatrati da je prazan kolosek je najmanje ograničen (na njega može stati bilo koji vagon), što možemo modelovati tako da na dno svih koloseka u početku postavimo fiktivnu vrednost $+\infty$.

Sve ovo omogućava da se napravi pohlepni algoritam koji izbegava proveru raznih mogućnosti nego do optimalnog rešenja (u ovom slučaju to je premeštanje svih vagona) tako što u svakom pojedinačnom koraku vagon postavlja na optimalno mesto (na drugi kraj pruge ako je to moguće, tj. na kolosek koji je u tom trenutku najviše ograničen).

```
#include <iostream>
#include <stack>
#include <queue>
#include <limits>
using namespace std;

int main() {
  // učitavamo vagone na ulazu i smeštamo ih u red iz kojeg će
  // izlaziti jedan po jedan
  queue<int> ulaz;
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    ulaz.push(x);
  }

  // tri pomoćna koloseka - jednostavnosti dalje analize radi na dno
  // svakog postavljamo vrednost beskonačno
  stack<int> s[3];
  for (int i = 0; i < 3; i++)
    s[i].push(numeric_limits<int>::max());

  // broj vagona koji treba da izađe
  int treba_da_izadje = 1;

  // da li je moguće sve vagone izvesti
  bool moze = true;
  while (treba_da_izadje <= n) {
```

```
  // ako se onaj koji treba da izađe nalazi na vrhu
  // nekog koloseka, njega izvodimo
  bool izasao_sa_pomocnog = false;
  for (int i = 0; i < 3; i++) {
    if (s[i].top() == treba_da_izadje) {
      cout << "Sa koloseka " << i << " izlazi "
           << treba_da_izadje << endl;
      s[i].pop(); treba_da_izadje++;
      izasao_sa_pomocnog = true;
    }
  }
  if (izasao_sa_pomocnog)
    continue;

  // uzimamo sledeći vagon sa ulaza
  int vagon = ulaz.front();
  ulaz.pop();

  // ako je on na redu, samo ga prevodimo na izlaz
  if (vagon == treba_da_izadje) {
    cout << "Sa ulaza izlazi " << treba_da_izadje << endl;
    treba_da_izadje++;
    continue;
  }

  // vagon stavljamo na onaj pomoćni kolosek na koji može
  // da stane, ali tako da je na vrhu tog koloseka najmanji
  // mogući broj
  int min = -1;
  for (int i = 0; i < 3; i++) {
    if (vagon < s[i].top() &&
        (min == -1 || s[i].top() < s[min].top()))
      min = i;
  }

  // ako nismo našli pomoćni kolosek,ređanje vagona
  // nije moguce
  if (min == -1) {
    moze = false;
    break;
  }

  // postavljamo vagon na odabrani pomoćni kolosek
  cout << "Sa ulaza vagon " << vagon
       << " ide na kolosek " << min << endl;
  s[min].push(vagon);
}

if (moze)
  cout << "moze" << endl;
```

```
  else
    cout << "ne moze" << endl;

  return 0;
}
```

Permutacija niske bez istih susednih elemenata

Problem: Naći neku permutaciju niske sastavljene od malih slova engleske abecede u kojoj nikoja dva susedna elementa nisu jednaka ili odrediti da takva permutacija ne postoji. Na primer, ako niska `aabbcc`, se može permutovati u `abacbc`, dok se niska `aabbbb` ne može permutovati. Dokazati korektnost algoritma.

I ovaj zadatak se može rešiti gramzivim pristupom. Naime, nakon popunjavanja određenog dela niske, na sledeće mesto možemo staviti karakter iz preostalog skupa karaktera, različit od poslednjeg postavljenog karaktera, koji se najčešće javlja u tom preostalom skupu karaktera.

Razmotrimo kako efikasno možemo implementirati prethodni gramzivi pristup. Pošto je u svakom koraku potrebno da određujemo karakter sa najvećom frekvencijom, zgodno je održavati red sa prioritetom u kom su karakteri poređani opadajuće po frekvencijama. Nakon uzimanja karaktera sa najvećom frekvencijom i postavljanja u nisku, na tekuće mesto, taj karakter možemo izbaciti iz reda sa prioritetom, jer je na sledećoj poziciji zabranjeno njegovo pojavljivanje. Međutim, nakon postavljanja sledećeg karaktera, prethodni karakter ponovo postaje kandidat, tako da ga u tom trenutku vraćamo nazad u red, ali sa frekvencijom umanjenom za jedan.

```
bool permutacija(string& s) {
  priority_queue<pair<int, char>> red;
  int frekvencije[26] = {0};
  for (char c : s)
    frekvencije[c - 'a']++;
  for (int i = 0; i < 26; i++)
    if (frekvencije[i] > 0)
      red.push(make_pair(frekvencije[i], 'a' + i));

  int preostalo_prethodnih;
  for (int i = 0; i < s.size(); i++) {
    if (red.empty()) return false;
    s[i] = red.top().second;
    int preostalo_trenutnih = red.top().first - 1;
    red.pop();
    if (i > 0 && preostalo_prethodnih > 0)
      red.push(make_pair(preostalo_prethodnih, s[i-1]));
    preostalo_prethodnih = preostalo_trenutnih;
  }
```

```
  return true;
}
```

Formiranje brojeva najmanjeg zbira

Problem: Od datog niza od n cifara (od 0 do 9) odrediti minimalnu moguću sumu dva broja formirana od cifara datog niza. Sve cifre niza moraju biti iskorišćene onoliko puta koliko se pojavljuju. Na primer, za $5, 3, 0, 7, 4$ dobija se 82 (tj. $35 + 047$). Dokazati korektnost.

Da bi zbir bio što manji, potrebno je da oba broja budu što kraća i da počinju što manjim ciframa. Kada odredimo najmanji zbir dobijen pomoćnu cifara različitih od nule, sve nule možemo dopisati na početak bilo kog od brojeva (ili ih ravnomerno rasporediti na oba broja) i zbir se neće promeniti. Ako bi se neka nula javljala na poziciji koja nije na početku broja, tada bi broj počinjao nekom cifrom koja nije nula i razmenom te cifre i nule dobio bi se manji broj, pa time i manji zbir. Dakle, sve nule možemo nadalje eliminisati iz razmatranja. Preostale cifre moramo ravnomerno raspoređivati u oba broja i ređati ih tako da su u svakom broju cifre opadajuće. Sve raspoložive cifre ćemo sortirati neopadajuće, održavaćemo tekuća dva broja i svaku narednu cifru ćemo dodavati na kraj manjeg od dva broja (ako su jednaki, svejedno je kojem će se cifra dodati). Naime, neka su tekući brojevi a i b, neka je $a \leq b$ i neka je tekuća cifra c. Dodavanjem cifre c na kraj manjeg broja a dobijamo zbir $10a + c + b$, a dodavanjem cifre c na kraj većeg broja b dobijamo zbir $10b + c + a$. Drugi zbir je uvek veći ili jednaki od prvog, jer je njihova razlika uvek nenegativna. Naime $(10b + c + a) - (10a + c + b) = 9(b - a) \geq 0$, jer je $b - a \geq 0$. Recimo još i da se u slučaju dodavanje cifre c nekom od brojeva na mesto koje nije poslednje daje zbir koji je veći ili jednak. Naime, pošto su cifre uređene neopadajuće važi da su sve cifre u broju manje ili jednake c. Ako poredimo broj u kome je cifra c dopisana na kraj i broj u kojem je cifra c dopisana negde u sredinu ili na početak, drugi će biti veći ili jednak od prvog, jer imaju isto cifara, a drugi će leksikografski biti veći od prvog.

```
int minZbir(int cifre[], int n) {
   sort(cifre, next(cifre, n));
   int a = 0, b = 0;
   for (int i = 0; i < n; i++) {
      a = 10 * a + cifre[i];
      if (a > b)
        swap(a, b);
   }
   return a + b;
}
```

Recimo i da bi dodavanje cifara naizmenično na kraj jednog, pa na kraj drugog broja takođe daje optimalno rešenje. Naime, svako dodavanje cifre na kraj

broja koji je manji ili jednak od drugog čini da taj broj postane veći ili jednak od drugog (zašto?).

Najveći broj sa cik-cak parnošću cifara

Problem: Od cifara datog broja (zadatog kao niska karaktera) sastaviti što je moguće veći broj, tako da nikoje dve susedne cifre tog broja nisu iste parnosti. Dokazati korektnost.

Najmanji broj dobijen konkatenacijom brojeva

Problem: Dato je n brojeva. Odredi najmanji broj koji se može dobiti njihovom konkatenacijom. Dokazati korektnost.

Matematičko predznanje

Matematička indukcija

Neka je $\mathbb{N} = \{1, 2, ...\}$ skup prirodnih brojeva. Pretpostavimo da treba dokazati da je tvrđenje $T(n)$ tačno za svaki prirodan broj $n \in \mathbb{N}$. Umesto da se ovo tvrđenje "napada" direktno, dovoljno je dokazati sledeća dva tvrđenja:

- $T(n)$ je tačno za $n = 1$, i
- za svako $n > 1$ važi: ako je tačno $T(n-1)$, onda je tačno i $T(n)$.

Ova činjenica je takozvani *princip matematičke indukcije*, i posledica je definicije skupa prirodnih brojeva: $1 \in \mathbb{N}$, a ako za neko $n > 1$ važi $n-1 \in \mathbb{N}$ onda $n \in \mathbb{N}$.

U praksi se obično prvo tvrđenje, tzv. *baza indukcije* lako dokazuje. Dokaz drugog tvrđenja olakšan je pretpostavkom da je tačno $T(n-1)$, odnosno *induktivnom hipotezom*. Drugim rečima, dovoljno je tvrđenje svesti na slučaj kad je n umanjeno za jedan.

Princip matematičke indukcije često se koristi u nešto promenjenim oblicima, koji su neposredna posledica osnovnog oblika.

Teorema: Ako je tačno tvrđenje $P(1)$ i za svako $n > 1$ iz pretpostavke da je $P(k)$ tačno za svako $k < n$ sledi tačnost $P(n)$, onda je $P(n)$ tačno za svako $n \in \mathbb{N}$.

Ovo je tzv. *potpuna indukcija*, a dobija se od osnovne varijante primenom na tvrđenje $T(n) = \bigwedge_{k=1}^{n} P(k)$.

Teorema[Regresivna indukcija]: Neka je $a_1, a_2, ...$ rastući niz prirodnih brojeva. Ako je tvrđenje $P(n)$ tačno za $n = a_k$, $k = 1, 2, ...$ i za svako $n \geq 2$ iz

pretpostavke da je tačno $P(n)$ sledi da je tačno $P(n-1)$, tada je $P(n)$ tačno za svako $n \in \mathbb{N}$.

Prvi deo pretpostavke, da je $P(a_k)$ tačno za $k \geq 1$ dokazuje se indukcijom ($P(1)$, i ako $P(a_k)$ onda $P(a_{k+1})$, $k \geq 1$). Dakle, najpre se dokazuje da postoje proizvoljno veliki brojevi n takvi da je $P(n)$ tačno, a onda da je $P(n)$ tačno i za sve "izostavljene" brojeve. Ovo je tzv. *regresivna indukcija.*

Prilikom izvođenja dokaza matematičkom indukcijom česta je greška da se kao očigledna činjenica iskoristi tvrđenje koje je blisko dokazivanom. Time se faktički pojačava induktivna hipoteza, ali se ne dokazuje da iz tačnosti pojačane hipoteze za n sledi njena tačnost za $n+1$.

Izračunavanje suma

Suma prvih n prirodnih brojeva

Označimo sa $S_n = 1 + 2 + ... + n = \sum_{k=1}^{n}$. Preuredimo elemente ove sume na sledeći način:

$S_n = 1 + \qquad 2 + \qquad 3 + ... + n$

$S_n = n + (n-1) + (n-2) + ... + 1$

Sabiranjem ove dve sume i grupisanjem sabiraka dobijamo:

$2S_n = (1+n) + (2+n-1) + (3+n-2) + ... + (n+1) = n(n+1)$

Stoga važi $S_n = \frac{n(n+1)}{2}$.

Gaus je još kao dete izračunao zbir brojeva od 1 do 100 tako što je primetio da se među njima nalazi 50 parova brojeva čiji je zbir 101, tako da se određivanje zbira prirodnih brojeva na taj način pripisuje Gausu.

Do ove formule smo mogli doći i posredstvom naredne vizuelizacije gde je u pravougaonik dimenzija $n(n+1)$ smešteno $1+2+...+n = S_n$ pluseva i isto toliko zvezdica. Odavde dobijamo da je $2S_n = n(n+1)$, odnosno da je $S_n = \frac{n(n+1)}{2}$.

```
+ + + +
+ + + *
+ + * *
+ * * *
* * * *
```

Do ove formule mogli smo doći i na treći način. Krenimo od razvoja kvadrata binoma $k-1$:

$$(k-1)^2 = k^2 - 2k + 1$$

Zamenom redosleda članova sa leve i sa desne strane dobijamo:

$$k^2 - (k-1)^2 = 2k - 1$$

Sumiranjem obe strane za $k = 1, 2, \ldots, n$ dobijamo:

$$\sum_{k=1}^{n} (k^2 - (k-1)^2) = 2\sum_{k=1}^{n} k - \sum_{k=1}^{n} 1$$

Skoro svi članovi sume sa leve strane će se pokratiti, ostaje samo n^2, dok je desna strana jednaka $2S_n - n$, odakle dobijamo da je $2S_n = n^2 + n$, odnosno da je $S_n = \frac{n(n+1)}{2}$. Ova tehnika je važna jer na analogan način možemo za proizvoljno k izračunati sumu k-tih stepena prvih n brojeva.

Suma kvadrata prvih n prirodnih brojeva

U duhu prethodne ideje, krenimo od razvoja kuba binoma $k - 1$:

$$(k-1)^3 = k^3 - 3k^2 + 3k - 1$$

Pogodnim preuređenjem članova dobijamo:

$$k^3 - (k-1)^3 = 3k^2 - 3k + 1$$

Kao i malopre, sumiramo obe strane po k, gde vrednost za k ide od 1 do n. Na taj način sa leve strane ostaje samo član n^3:

$$n^3 = 3\Big(\sum_{k=1}^{n} k^2\Big) - 3\sum_{k=1}^{n} k + \sum_{k=1}^{n} 1$$

$$n^3 = 3\Big(\sum_{k=1}^{n} k^2\Big) - 3\frac{n(n+1)}{2} + n$$

$$3\Big(\sum_{k=1}^{n} k^2\Big) = n^3 + 3\frac{n(n+1)}{2} - n$$

$$\sum_{k=1}^{n} k^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n = \frac{n(n+1)(2n+1)}{6}$$

Suma k-tih stepena prvih n prirodnih brojeva – *stepene sume*

Ako sa $S_k(n)$ označimo opštiju sumu k-tih stepena prvih n prirodnih brojeva, $S_k(n) = \sum_{i=1}^{n} i^k$, za izračunavanje suma $S_k(n)$ može se iskoristiti rekurentna relacija

$$S_k(n) = -\frac{1}{k+1}\sum_{j=0}^{k-1}\binom{k+1}{j}S_j(n) + \frac{1}{k+1}((n+1)^{k+1} - 1).$$

u kojoj je vrednost $S_k(n)$ izražena preko svih prethodnih vrednosti $S_j(n)$, $j = 0, \ldots, k-1$.

Ova jednakost dobija se na sledeći način:

$$(i+1)^{k+1} = \sum_{j=0}^{k+1}\binom{k+1}{j}i^j = \sum_{j=0}^{k-1}\binom{k+1}{j}i^j + (k+1)i^k + i^{k+1}$$

te ako obe strane podelimo sa $(k+1)$ i izrazimo i^k dobijamo:

$$i^k = -\frac{1}{k+1}\sum_{j=0}^{k-1}\binom{k+1}{j}i^j + \frac{1}{k+1}((i+1)^{k+1} - i^{k+1}),$$

Sumiranjem ovih jednakosti za $i = 1, 2, \ldots, n$ dobijamo traženi izraz za $S_k(n)$.

Polazeći od $S_0(n) = n$, $S_1(n) = n(n+1)/2$, za $k = 2$ se dobija

$$\begin{aligned} S_2(n) = \sum_{i=1}^{n} i^2 &= -\frac{1}{3}(S_0(n) + 3S_1(n)) + \frac{1}{3}((n+1)^3 - 1) = \\ &= \frac{1}{3}\left(-n - 3\frac{n(n+1)}{2} + n^3 + 3n^2 + 3n\right) = \frac{1}{6}(2n^3 + 3n^2 + n) = \\ &= \frac{1}{6}n(n+1)(2n+1). \end{aligned}$$

Aritmetički i geometrijski niz

Aritmetički niz tj. *aritmetička progresija* je niz brojeva u kome je razlika svakog člana i njemu prethodnog člana konstantna, odnosno ovaj niz je oblika $a, a + d, a + 2 \cdot d, \ldots, a + n \cdot d$. Jednostavnom analizom može se zaključiti da je i-ti član aritmetičke progresije jednak $a_i = a + (i-1) \cdot d$.

Razmotrimo problem određivanja sume svih elemenata aritmetičkog niza $a_1 + a_2 + \ldots + a_n$. Obeležimo traženu sumu sa S. Primenom formule za a_i dobijamo da je:

$S = a + (a + d) + ... + (a + (n-2) \cdot d) + (a + (n-1) \cdot d)$

Ako traženu sumu napišemo u direktnom i inverznom poretku dobijamo:

$S = a + (a + d) + ... + (a + (n-2) \cdot d) + (a + (n-1) \cdot d)$

$S = (a + (n-1) \cdot d) + (a + (n-2) \cdot d) + ... + (a + d) + a$

Sabiranjem ove dve jednakosti član po član (sabiramo prve članove, druge članove, itd.) dobijamo:

$2 \cdot S = (2 \cdot a + (n-1) \cdot d) + (2 \cdot a + (n-1) \cdot d) + ... + (2 \cdot a + (n-1) \cdot d) + (2 \cdot a + (n-1) \cdot d)$

odnosno $2 \cdot S = n \cdot (2 \cdot a + (n-1) \cdot d)$. Iz poslednje jednakosti dobijamo $S = \frac{n \cdot (2 \cdot a + (n-1) \cdot d)}{2}$. Primetimo da je proizvod $n \cdot (2 \cdot a + (n-1) \cdot d)$ deljiv sa 2 jer je $2 \cdot a$ parno i paran je broj n ili $n-1$, pa je prema tome jedan od faktora posmatranog proizvoda paran i moguće je upotrebiti celobrojno deljenje brojem 2.

Drugi način da dođemo do rezultata je da primetimo da je $S = n \cdot a + d \cdot (1 + 2 + ... + n - 1)$, i da se pozovemo na Gausovu formulo koja kaže da je suma prirodnih brojeva od 1 do $n-1$ jednaka $\frac{(n-1)n}{2}$ (dok je suma brojeva od 1 do n jednaka $\frac{n(n+1)}{2}$).

Geometrijski niz tj. *geometrijska progresija* je niz brojeva u kome je količnik svakog člana i njemu prethodnog člana konstantan, odnosno ovaj niz je oblika $a, a \cdot q, a \cdot q^2, ..., a \cdot q^{n-1}$. Jednostavnom analizom može se utvrditi da je i-ti član geometrijskog niza jednak $a_i = a \cdot q^{i-1}$.

Razmotrimo problem određivanja sume $a_1 + a_2 + ... + a_{n-1} + a_n$. Obeležimo traženu sumu sa S. Primenom formule za a_i dobijamo da je $S = a + a \cdot q^1 + a \cdot q^2 + ... + a \cdot q^{n-2} + a \cdot q^{n-1}$.

Ako levu i desnu stranu prethodne jednakosti pomnožimo sa $1-q$ (napomenimo da je $1 - q \neq 0$) dobijamo jednakost:

$S \cdot (1-q) = a \cdot (1-q) + a \cdot q \cdot (1-q) + a \cdot q^2 \cdot (1-q) + ... + a \cdot q^{n-2} \cdot (1-q) + a \cdot q^{n-1} \cdot (1-q)$

Nakon izvršenog množenja na desnoj strani jednakosti, dobijamo narednu jednakost:

$S \cdot (1-q) = a - a \cdot q + a \cdot q - a \cdot q^2 + a \cdot q^2 - a \cdot q^3 + ... + a \cdot q^{n-2} - a \cdot q^{n-1} + a \cdot q^{n-1} - a \cdot q^n$

Sređivanjem poslednjeg izraza dobijamo $S \cdot (1-q) = a - a \cdot q^n$. Prema tome, važi $S = a \frac{1-q^n}{1-q}$.

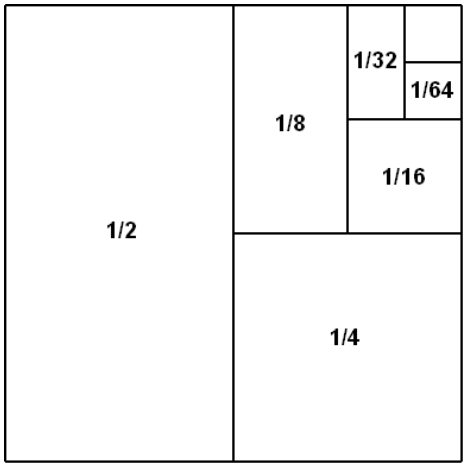
Ukoliko bismo umesto konačne sume $S = \sum_{i=0}^{n-1} aq$ razmatrali geometrijski red $S' = \sum_{i=0}^{\infty} aq$ za $q < 1$ dobili bismo da je:

$S' = a \lim_{n \to \infty} \frac{1-q^n}{1-q} = a \frac{1}{1-q}$.

U specijalnom slučaju kada je $a = 1, q = 1/2$ dobijamo:

$$\sum_{i=1}^{\infty}\left(\frac{1}{2}\right)^i = \frac{1}{2}+\frac{1}{4}+\frac{1}{8}+...=\frac{1}{1-1/2}-1=2-1=1$$

Na slici dat je vizuelni prikaz ove sume.



Polazeći od izraza za sumu geometrijske progresije može se izračunati suma

$$G(n)=\sum_{i=1}^{n} i2^i$$

. Zaista, diferenciranjem, pa množenjem sa x jednakosti

$$\sum_{i=0}^{n} x^i = (x^{n+1}-1)/(x-1)$$

dobija se

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

Odatle se za $x=2$ dobija $G(n)=\sum_{i=1}^{n} i2^i=(n-1)2^{n+1}+2$.

Slično, svođenjem na prethodnu sumu može se izračunati suma

$$H(n)=\sum_{i=1}^{n} i2^{n-i}.$$

Zaista, smenom indeksa sumiranja $j=n-i$, $i=n-j$, pri čemu j prolazi skup vrednosti $n-1, n-2, ..., n-n=0$, dobija se:

$$H(n)=\sum_{j=0}^{n-1}(n-j)2^j = n\sum_{j=0}^{n-1}2^j-\sum_{j=0}^{n-1}j2^j = nF(n-1)-G(n-1)=$$
$$=n(2^n-1)-(n-2)2^n-2=2^{n+1}-n-2.$$

###Ocenjivanje suma pomoću integrala

U slučaju kad je teško izračunati sumu $\sum_{i=1}^{n} f(i)$, a funkcija $f(x)$ od realnog argumenta je monotono neopadajuća neprekidna funkcija za $x \geq 1$, tada se sumiranjem levih strana za $i = 1, 2, \ldots, n$, a desnih strana za $i = 1, 2, \ldots, n-1$ nejednakosti:

$$f(i) \leq \int_{i}^{i+1} f(x)\,dx \leq f(i+1), \quad 1 \leq i \leq n,$$

dobijaju granice intervala u kome leži suma $\sum_{i=1}^{n} f(i)$:

$$\int_{1}^{n} f(x)\,dx + f(1) \leq \sum_{i=1}^{n} f(i) \leq \int_{1}^{n+1} f(x)\,dx.$$

videti sliku 12.

Ako je pak funkcija $f(x)$ monotono nerastuća, onda je funkcija $-f(x)$ neopadajuća, pa se primenom ovih nejednakosti na $-f(x)$ dobija da u ovim nejednakostima samo treba promeniti znak "$\leq$" u znak "$\geq$".

Primer: Razmotrimo problem računanja sume $S_k(n) = \sum_{i=1}^{n} i^k$ za $k > 0$. U ovom slučaju funkcija $f(x) = x^k$ je monotono neopadajuća, pa se dobija procena:

$$\int_{1}^{n} f(x)\,dx + f(1) \leq \sum_{i=1}^{n} f(i) \leq \int_{1}^{n+1} f(x)\,dx$$

odnosno:

$$\int_{1}^{n} x^k\,dx + 1 \leq \sum_{i=1}^{n} i^k \leq \int_{1}^{n+1} x^k\,dx$$

Za $k \neq -1$ je:

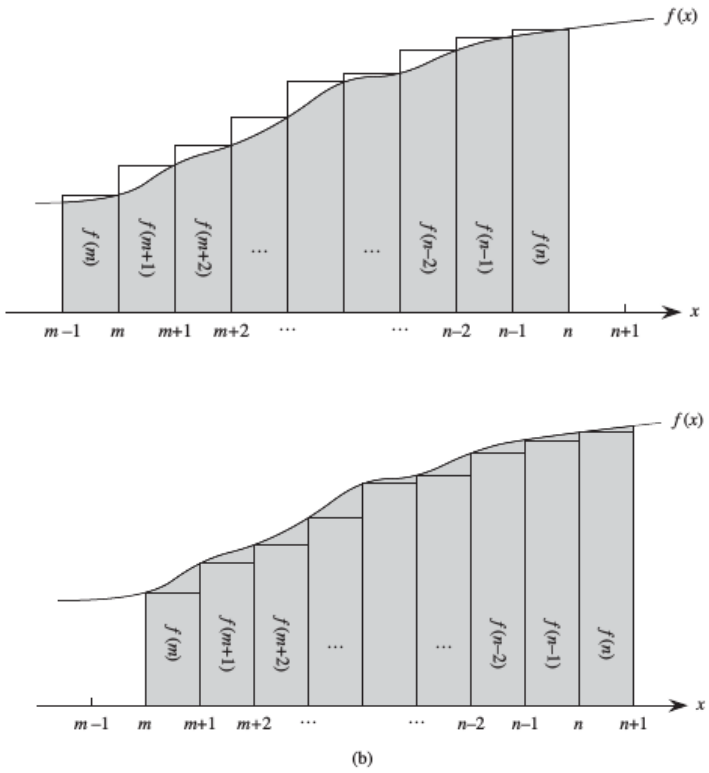
$$\int x^k\,dx = \frac{x^{k+1}}{k+1} + C$$

Odavde za $k \neq -1$ dobijamo narednu ocenu tražene sume:

$$\frac{n^{k+1}-1}{k+1} + 1 \leq \sum_{i=1}^{n} i^k \leq \frac{(n+1)^{k+1}-1}{k+1}$$

odnosno:

$$\frac{n^{k+1}}{k+1} \leq \frac{n^{k+1}+k}{k+1} \leq \sum_{i=1}^{n} i^k \leq \frac{(n+1)^{k+1}-1}{k+1}$$



Slika 12: Aproksimacija sume $\sum_{k=m}^{n} f(k)$ integralima. Ukupna površina svih pravougaonika predstavlja vrednost sume, dok je vrednost integrala jedanaka osenčenoj površini ispod krive. Obratiti pažnju da je horizontalna stranica svakog od pravougaonika dužine 1.

tj:

$$S_k(n) = \sum_{i=1}^{n} i^k = \Theta(n^{k+1})$$

Specijalno, za $k = 2$ dobijamo:

$$\sum_{i=1}^{n} i^2 = \Theta(n^3),$$

a za $k = 1/2$:

$$\sum_{i=1}^{n} \sqrt{i} = \Theta(n^{3/2}) = \Theta(n\sqrt{n}).$$

Ukoliko je $k < 0$, funkcija $f(x) = x^k$ je monotono opadajuća (za $x > 0$) pa za $k \neq -1$ važi naredna ocena:

$$\frac{(n+1)^{k+1}}{k+1} - 1 \leq \sum_{i=1}^{n} i^k \leq \frac{n^{k+1}}{k+1}$$

Prema tome, za $-1 < k < 0$ je $S_k(n) = \Theta(n^{k+1})$, a za $k < -1$ je $S_k(n) = \Theta(1)$. U slučaju $k = -1$ važi da je:

$$\int \frac{1}{x}\, dx = \ln x + C$$

i:

$$\ln(n+1) \leq \sum_{i=1}^{n} \frac{1}{i} \leq \ln(n) + 1$$

te važi:

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$$

Primer: Razmotrimo problem ocene vrednosti $\ln(n!) = \sum_{i=1}^{n} \ln i$. Funkcija $f(x) = \ln x$ je monotono neopadajuća te možemo koristiti narednu procenu:

$$\int_1^n \ln x\, dx + \ln 1 \leq \sum_{i=1}^{n} \ln i \leq \int_1^{n+1} \ln x\, dx$$

Pošto važi da je $\ln 1 = 0$ i da je $\int \ln x = x \ln x - x$, dobijamo ocenu:

$$(x \ln x - x)|_1^n \leq \ln n! \leq (x \ln x - x)|_1^{n+1}$$

odnosno:

$$n \ln n - n + 1 \leq \ln n! \leq (n+1)\ln(n+1) - (n+1) + 1$$

Iz ove procene, ako svaku stranu nejednakosti uzmemo kao eksponent broja e dobijamo:

$$e\left(\frac{n}{e}\right)^n \leq n! \leq e\left(\frac{n+1}{e}\right)^{n+1} = e\left(\frac{n}{e}\right)^{n+1}\left(1+\frac{1}{n}\right)^{n+1} \leq e\left(\frac{n}{e}\right)^n (n+1)$$

Precizniji izraz za $n!$:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

zove se *Stirlingova formula*.

Analiza algoritama

Definicija: Neka su f i g pozitivne funkcije od argumenta n iz skupa $\mathbf{N}$ prirodnih brojeva. Kaže se da je $g(n) = O(f(n))$ ako postoje pozitivne konstante c i N, takve da za svako $n > N$ važi $g(n) \leq cf(n)$.

Oznaka $O(f(n))$ se u stvari odnosi na *klasu funkcija*, a jednakost $g(n) = O(f(n))$ je uobičajena oznaka za *inkluziju* $g(n) \in O(f(n))$. Jasno je da je funkcija f samo neka vrsta *gornje granice* za funkciju g. Na primer, pored jednakosti $5n^2+15 = O(n^2)$ (jer je $5n^2+15 \leq 6n^2$ za $n \geq 4$) važi i jednakost $5n^2+15 = O(n^3)$ (jer je $5n^2+15 \leq 6n^3$ za $n \geq 4$). Ova notacija omogućuje ignorisanje multiplikativnih konstanti: umesto $O(5n+4)$ može se pisati $O(n)$ jer klase $O(5n+4)$ i $O(n)$ sadrže iste funkcije. Jednakost $O(5n+4) = O(n)$ je u stvari jednakost dve klase funkcija. Slično, u izrazu $O(\log n)$ osnova logaritma nije bitna, jer se logaritmi za različite osnove razlikuju za multiplikativnu konstantu:

$$\log_a n = \log_a(b^{\log_b n}) = \log_b n \cdot \log_a b.$$

Specijalno, $O(1)$ je oznaka za *klasu ograničenih funkcija*.

Teorema: Svaka eksponencijalna funkcija sa osnovom većom od 1 raste brže od svakog polinoma. Dakle, ako je $f(n)$ monotono rastuća funkcija koja nije ograničena, $a > 1$ i $c > 0$, onda je:

$$f(n)^c = O(a^{f(n)}).$$

Specijalno, za $f(n) = n$ dobija se $n^c = O(a^n)$, a za $f(n) = \log_a n$ dobija se jednakost $(\log_a n)^c = O(a^{log_a n}) = O(n)$, tj. proizvoljan stepen logaritamske funkcije raste sporije od linearne funkcije.

Lako se pokazuje da se O-izrazi mogu sabirati i množiti:

$$\begin{aligned} O(f(n)) + O(g(n)) &= O(f(n) + g(n)), \\ O(f(n))O(g(n)) &= O(f(n)g(n)). \end{aligned}$$

Na primer, druga od ovih jednakosti može se iskazati na sledeći način: ako je neka funkcija $h(n)$ jednaka proizvodu proizvoljne funkcije $r(n)$ iz klase $O(f(n))$ i proizvoljne funkcije $s(n)$ iz klase $O(g(n))$, onda $h(n) = r(n)s(n)$ pripada klasi $O(f(n)g(n)))$.

Pokažimo da važe ove dve jednakosti. Ako je $r(n) = O(f(n))$ i $s(n) = O(g(n))$ onda postoje pozitivni N_1, N_2, c_1 i c_2 takvi da za $n > N_1$ važi $r(n) \leq c_1 f(n)$ i za $n > N_2$ važi $s(n) \leq c_2 g(n)$. Međutim, za $N = \max\{N_1, N_2\}$ i $c = \max\{c_1, c_2, c_1 c_2\}$ važi:

- $r(n) + s(n) \leq c(f(n) + g(n))$ i
- $r(n)s(n) \leq cf(n)g(n)$,

tj. $r(n) + s(n) = O(f(n) + g(n))$ i $r(n)s(n) = O(f(n)g(n))$. Međutim, O-izrazi odgovaraju relaciji $\leq$, pa se ne mogu oduzimati i deliti. Tako npr. iz $f(n) = O(r(n))$ i $g(n) = O(s(n))$ ne sledi da je $f(n) - g(n) = O(r(n) - s(n))$.

Drugi problem u vezi sa složenošću algoritma je pitanje donje granice za potreban broj računskih operacija. Dok se gornja granica odnosi na *konkretan algoritam*, donja granica složenosti odnosi se na *proizvoljan algoritam iz neke određene klase*. Zbog toga ocena donje granice zahteva posebne postupke analize. Za funkciju $g(n)$ kaže se da je *asimptotska donja granica* funkcije $T(n)$ i piše se $T(n) = \Omega(g(n))$, ako postoje pozitivne konstante c i N, takve da za svako $n > N$ važi $T(n) > cg(n)$. Tako je na primer $n^2 = \Omega(n^2 - 100)$, i $n = \Omega(n^{0.9})$. Vidi se da simbol Ω odgovara relaciji $\geq$. Ako za dve funkcije $f(n)$ i $g(n)$ istovremeno važi i $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$, onda one imaju iste asimptotske brzine rasta, što se označava sa $f(n) = \Theta(g(n))$. Tako je na primer $5n \log_2 n - 10 = \Theta(n \log n)$, pri čemu je u poslednjem izrazu osnova logaritma nebitna.

Na kraju, činjenica da je $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ označava se sa $f(n) = o(g(n))$. Na primer, očigledno je $\frac{n}{\log_2 n} = o(n)$, a jednakost $\frac{n}{10} = o(n)$ nije tačna. Odnos između stepene i eksponencijalne funkcije može se precizirati: ako je $f(n)$ monotono rastuća funkcija koja nije ograničena, $a > 1$ i $c > 0$, onda je

$$f(n)^c = o(a^{f(n)})$$

##Diferencne jednačine

Ako za članove niza F_n, $n = 1, 2, ...$, važi jednakost

$$F_n = f(F_{n-1}, F_{n-2}, \dots, F_1, n),$$

onda se kaže da niz F_n zadovoljava tu *diferencnu jednačinu* (ili *rekurentnu relaciju*). Specijalno, ako se za neko $k \geq 1$ član F_n izražava preko k prethodnih članova niza,

$$F_n = f(F_{n-1}, F_{n-2}, \dots, F_{n-k}, n),$$

onda je k *red* te diferencne jednačine. Matematičkom indukcijom se neposredno dokazuje da je drugom diferencnom jednačinom i sa prvih k članova F_1, F_2, ...,

F_k niz F_n jednoznačno određen. Drugim rečima, ako neki niz G_n zadovoljava istu diferencnu jednačinu $G_n = f(G_{n-1}, G_{n-2}, \ldots, G_{n-k}, n)$, i važi $F_i = G_i$ za $i = 1, 2, \ldots, k$, onda za svako $n \geq 1$ važi $F_n = G_n$.

Jedna od najpoznatijih diferencnih jednačina je ona koja definiše Fibonačijev niz,

$$F_n = F_{n-1} + F_{n-2}, \quad F_1 = F_2 = 1. \tag{1}$$

Da bi se na osnovu nje izračunala vrednost F_n potrebno je izvršiti $n - 2$ koraka (sabiranja), pa je ovakav način izračunavanja F_n nepraktičan za velike n. Taj problem je zajednički za sve nizove zadate diferencnim jednačinama. Izraz koji omogućuje direktno izračunavanje proizvoljnog člana niza (dakle ne preko prethodnih) zove se *rešenje diferencne jednačine*. Na diferencne jednačine se često nailazi pri analizi algoritama. Zbog toga ćemo razmotriti nekoliko metoda za njihovo rešavanje.

###Dokazivanje pretpostavljenog rešenja diferencne jednačine

Često se diferencne jednačine rešavaju tako što se pretpostavi oblik rešenja, ili čak tačno rešenje, posle čega se posle eventualnog preciziranja rešenja indukcijom dokazuje da to jeste rešenje. Pri tome je često drugi deo posla, dokazivanje, lakši od prvog — pogađanja oblika rešenja.

Primer: Posmatrajmo diferencnu jednačinu

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

Ova diferencna jednačina definiše vrednosti $T(n)$ samo za indekse oblika $n = 2^m$, $m = 1, 2, \ldots$. Postavimo sebi skromniji cilj — pronalaženje neke gornje granice za $T(n)$, odnosno funkcije $f(n)$ koja zadovoljava uslov $T(n) \leq f(n)$ (za indekse n koji su stepen dvojke, što će se nadalje podrazumevati), ali tako da procena bude dovoljno dobra. Pokušajmo najpre sa funkcijom $f(n) = n^2$. Očigledno je $T(2) = 1 < f(2) = 4$. Ako pretpostavimo da je $T(i) \leq i^2$ za $i \leq n$ (induktivna hipoteza), onda je

$$T(2n) = 2T(n) + 2n - 1 \leq 2n^2 + 2n - 1 = 4n^2 - \big(2n(n-1) + 1\big) < 4n^2,$$

pa je $f(n)$ gornja granica za $T(n)$ za sve (dozvoljene) vrednosti n. Međutim, iz dokaza se vidi da je dobijena granica gruba: u odnosu na član $4n^2$ odbačen je član približno jednak $2n^2$ za velike n. S druge strane, ako se pretpostavi da je gornja granica linearna funkcija $f(i) = ci$ za $i \leq n$, onda bi trebalo dokazati da ona važi i za $i = 2n$:

$$T(2n) = 2T(n) + 2n - 1 \leq 2cn + 2n - 1.$$

Da bi poslednji izraz bio manji od $c \cdot 2n$, moralo bi da važi $2n - 1 \leq 0$, što je nemoguće za $n \geq 1$. Zaključujemo da $f(n) = cn$ ne može da bude gornja granica za $T(n)$, bez obzira na vrednost konstante c. Trebalo bi pokušati sa nekim izrazom koji je po asimptotskoj brzini rasta između n i n^2. Ispostavlja se da je

dobra granica zadata funkcijom $f(n) = n \log_2 n$: $T(2) = 1 \leq f(2) = 2 \log_2 2 = 2$, a ako je $T(i) \leq f(i)$ tačno za $i \leq n$, onda je

$$T(2n) = 2T(n) + 2n - 1 \leq 2n \log_2 n + 2n - 1 = 2n \log_2(2n) - 1 < 2n \log_2(2n).$$

Dakle, $T(n) \leq n \log_2 n$ je tačno za svako n oblika 2^k. Na osnovu činjenice da je u ovom dokazu pri prelasku sa n na $2n$ zanemaren član 1 u odnosu na $2n \log_2(2n)$, zaključuje se da je procena ovog puta dovoljno dobra.

"Diferencna nejednačina"

$$T(2n) \leq 2T(n) + 2n - 1, \quad T(2) = 1$$

definiše samo gornje granice za $T(n)$, ako je n stepen dvojke. Svaka gornja granica rešenja (2) je rešenje ove diferencne nejednačine — dokazi indukcijom su praktično identični. Ova diferencna nejednačina može se, kao i (2), proširiti tako da određuje gornju granicu za $T(n)$ za svaki prirodan broj n:

$$T(n) \leq 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1, \quad T(1) = 0.$$

Može se pokazati da za niz $T(n)$, koji zadovoljava ovaj uslov, nejednakost $T(n) \leq n \log_2 n$ važi za svako $n \geq 1$. Zaista, ona je tačna za $n = 1$, a iz pretpostavke da je ona tačna za brojeve manje od nekog prirodnog broja m sledi da je za $m = 2k$:

$$T(2k) \leq 2T(k) + 2k - 1 \leq 2k \log_2 k + 2k - 1 = 2k \log_2(2k) - 1 < (2k) \log_2(2k)$$

odnosno za $m = 2k + 1$:

$$T(2k+1) \leq 2T(k) + 2k \leq 2k \log_2 k + 2k = 2k \log_2(2k) < (2k+1) \log_2(2k+1),$$

tj. data nejednakost je tada tačna i za $n = m$. Dakle, ova nejednakost je dokazana indukcijom.

###Linearne diferencne jednačine

Razmotrićemo sada homogene linearne diferencne jednačine oblika

$$F(n) = a_1 F(n-1) + a_2 F(n-2) + \cdots + a_k F(n-k). \tag{3}$$

Rešenja se mogu tražiti u obliku:

$$F(n) = r^n, \tag{4}$$

gde je r pogodno izabrani (u opštem slučaju) *kompleksni* broj. Zamenom u (3) dobija se uslov koji r treba da zadovolji:

$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - \cdots - a_k = 0,$$

takozvana *karakteristična jednačina* linearne diferencne jednačine (3); polinom sa leve strane ove jednačine je *karakteristični polinom* ove linearne diferencne

jednačine. U slučaju kad su svi koreni karakterističnog polinoma $r_1, r_2, \ldots, r_k$ različiti, dobijamo k rešenja (3) oblika (4). Tada se svako rešenje može predstaviti u obliku:

$$F(n) = \sum_{i=1}^{k} c_i r_i^n$$

(ovo tvrđenje navodimo bez dokaza). Ako je prvih k članova niza $F(n)$ zadato, rešavanjem sistema od k linearnih jednačina dobijaju se koeficijenti c_i, $i = 1, 2, \ldots, k$, a time i traženo rešenje.

Primer: Opšti član niza zadatog diferencnom jednačinom

$$F_n = F_{n-1} + F_{n-2},$$

i početnim uslovima $F_1 = F_2 = 1$ (Fibonačijevog niza) može se odrediti na opisani način. Koreni karakteristične jednačine $r^2 - r - 1 = 0$ su $r_1 = (1+\sqrt{5})/2$ i $r_2 = (1-\sqrt{5})/2$, pa je rešenje ove diferencne jednačine oblika:

$$F_n = c_1 r_1^n + c_2 r_2^n.$$

Koeficijenti c_1 i c_2 određuju se iz uslova $F_1 = F_2 = 1$, ili nešto jednostavnije iz uslova $F_0 = F_2 - F_1 = 0$ i $F_1 = 1$:

$$\begin{aligned} c_1 \quad &+ c_2 \quad &= 0 \\ c_1 r_1 \quad &+ c_2 r_2 \quad &= 1. \end{aligned}$$

Rešenje ovog sistema jednačina je $c_1 = -c_2 = \frac{1}{\sqrt{5}}$, pa je opšti član Fibonačijevog niza zadat sa:

$$F_n = \frac{1}{\sqrt{5}} \Big(\Big(\frac{1+\sqrt{5}}{2} \Big)^n - \Big(\frac{1-\sqrt{5}}{2} \Big)^n \Big). \tag{5}$$

U opštem slučaju, ako su koreni karakteristične jednačine $r_1, r_2, \ldots, r_s$ višestrukosti redom $m_1, m_2, \ldots, m_s$ ($\sum m_i = k$), onda se svako rešenje jednačine (3) može predstaviti u obliku:

$$F(n) = \sum_{i=1}^{s} h_i(n),$$

gde je

$$h_i(n) = \left(C_{i0} + C_{i1} n + C_{i2} n^2 + \cdots + C_{i,m_i-1} n^{m_i-1} \right) r_i^n$$

(ovo tvrđenje takođe navodimo bez dokaza).

Primer: Karakteristični polinom diferencne jednačine

$$F(n) = 6F(n-1) - 12F(n-2) + 8F(n-3)$$

je $r^3 - 6r^2 + 12r - 8 = (r-2)^3$, sa trostrukim korenom $r_1 = 2$. Zbog toga su sva rešenja ove diferencne jednačine oblika $F(n) = (c_0 + c_1 n + c_2 n^2)2^n$, gde su c_0, c_1, c_2 konstante koje se mogu odrediti ako se znaju npr. tri prva člana niza $F(1)$, $F(2)$ i $F(3)$.

###Diferencne jednačine koje se rešavaju sumiranjem

Često se nailazi na linearnu nehomogenu diferencnu jednačinu oblika

$$F(n+1) - F(n) = f(n), \tag{6}$$

pri čemu $F(0)$ ima zadatu vrednost. Drugim rečima, razlika dva uzastopna člana traženog niza jednaka je datoj funkciji od indeksa n. Rešavanje ovakve diferencne jednačine svodi se na izračunavanje sume $\sum_{i=1}^{n-1} f(i)$. Zaista, ako u gornjoj jednakosti n zamenimo sa i i izvršimo sumiranje njene leve i desne strane po i u granicama od 0 do $n-1$, dobijamo

$$\sum_{i=0}^{n-1} f(i) = \sum_{i=0}^{n-1} F(i+1) - \sum_{i=0}^{n-1} F(i) = \sum_{i=1}^{n} F(i) - \sum_{i=0}^{n-1} F(i) = F(n) - F(0).$$

Prema tome, rešenje diferencne jednačine (6) dato je izrazom

$$F(n) = \sum_{i=0}^{n-1} f(i) + F(0). \tag{7}$$

Primer: Razmotrimo diferencnu jednačinu $F(n+1) = F(n)+2n+1$ sa početnim uslovom $F(0) = 0$. Njeno rešenje je

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

###Diferencne jednačine za algoritme zasnovane na razlaganju

Pretpostavimo da nam je cilj analiza algoritma A, pri čemu broj operacija $T(n)$ pri primeni algoritma A na ulaz veličine n (vremenska složenost) zadovoljava diferencnu jednačinu oblika

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k, \tag{8}$$

pri čemu je $a, b, c, k \geq 0$, $b \neq 0$, i zadata je vrednost $T(1)$. Ovakva jednačina dobija se za algoritam kod koga se obrada ulaza veličine n svodi na obradu a ulaza veličine n/b, posle čega je potrebno izvršiti još cn^k koraka da bi se od parcijalnih rešenja konstruisalo rešenje kompletnog ulaza veličine n. Jasno je zašto se za ovakve algoritme kaže da su tipa "zavadi pa vladaj", (eng. divide–and–conquer), kako se još zovu algoritmi zasnovani na razlaganju. Obično je u ovakvim diferencnim jednačinama b prirodan broj.

Teorema [Master teorema]: Asimptotsko ponašanje niza $T(n)$, rešenja diferencne jednačine (8) dato je jednakošću:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{za } a > b^k \\ O(n^k \log n) & \text{za } a = b^k \\ O(n^k) & \text{za } a < b^k \end{cases}. \tag{9}$$

Dokaz: Dokaz teoreme biće sproveden samo za podniz $n = b^m$, gde je m celi nenegativni broj. Pomnoživši obe strane jednakosti (8) sa a^{-m}/c, dobijamo diferencnu jednačinu tipa (6)

$$t_m = t_{m-1} + q^m, \quad t_0 = \frac{1}{c}\, T(1),$$

gde su uvedene oznake $t_m = \frac{1}{c} a^{-m} T(b^m)$ i $q = b^k/a$. Njeno rešenje je

$$t_m = t_0 + \sum_{i=1}^{m} q^i$$

Za $q \neq 1$ je $\sum_{i=1}^{m} q^i = (1 - q^{m+1})/(1 - q) - 1$, pa se asimptotsko ponašanje niza t_m opisuje sledećim jednakostima:

$$t_m = \begin{cases} O(m), & \text{za } q = 1 \\ O(1), & \text{za } 0 < q < 1 \\ O(q^m), & \text{za } q > 1 \end{cases}.$$

Pošto je $T(b^m) = ca^m t_m$, $n = b^m$, odnosno $m = \log_b n$, redom se za $0 < q < 1$ ($b^k < a$), $q = 1$ ($b^k = a$, odnosno $\log_b a = k$) i $q > 1$ ($b^k > a$) dobija

$$T(n) = \begin{cases} O(a^m) = O(b^{\log_b a^m}) = O(b^{m \log_b a}) = O(n^{\log_b a}) & \text{za } a > b^k \\ O(ma^m) = O(\log_b n \cdot n^{\log_b a}) = O(n^k \log n) & \text{za } a = b^k \\ O((aq)^m) = O(b^{mk}) = O(n^k) & \text{za } a < b^k \end{cases},$$

čime je tvrđenje teoreme dokazano.

Algoritmi ovog tipa imaju široku primenu zbog svoje efikasnosti, pa je korisno znati asimptotsko ponašanje rešenja diferencne jednačine (8).

###Potpuna rekurzija

Diferencna jednačina najopštijeg oblika () zove se *potpuna rekurzija* ili *diferencna jednačina sa potpunom istorijom.* Razmotrićemo dva primera ovakvih diferencnih jednačina.

Primer: Neka za $n \geq 2$ važi $T(n) = c + \sum_{i=1}^{n-1} T(i)$, pri čemu su c i $T(1)$ zadati. Osnovna ideja za rešavanje ovakvih problema je tzv. "eliminacija istorije" nalaženjem ekvivalentne diferencne jednačine sa "konačnom istorijom" (takve

kod koje se član niza izražava preko ograničenog broja prethodnih). Zamenom n sa $n-1$ u ovoj jednačini dobija se $T(n-1) = c + \sum_{i=1}^{n-2} T(i)$. Oduzimanjem druge od prve jednakosti postiže se željeni efekat:

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

(za $n \geq 2$), i konačno

$$T(n) = 2T(n-1) = 2^2 T(n-2) = \cdots = 2^{n-2} T(2) = 2^{n-2}(c + T(1)).$$

Primetimo da se jednakost $T(n) = 2T(n-1)$ logaritmovanjem i smenom $t_n = \log T(n)$ svodi na jednačinu $t_n = t_{n-1} + \log 2$, odnosno diferencnu jednačinu tipa (6).