Algoritmi i strukture podataka

Minimalni nivo

1. Induktivno-rekurzivna konstrukcija – pojam, ilustrovanje kroz primere

Induktivno-rekurzivni pristup podrazumeva svođenje početnog problema na problem (ili više problema) manje dimenzije, a istog oblika i dobijanja rešenja početnog problema pomoću rešenja tog manjeg problema. Za početne dimenzije problem se rešava direktno, bez daljeg svođenja. Ova konstrukcija je u uskoj vezi sa **matematičkom indukcijom**:

$$(P(0) \wedge (\forall n)(P(n) \Rightarrow P(n+1))) \Rightarrow (\forall n)(P(n))$$

Da bismo dokazali da svaki prirodan broj n ima svojstvo P(n) dovoljno je da dokažemo da za 0 važi to svojstvo i da ukoliko svojstvo važi za neki broj onda će važiti i za njegov sledbenik. Prvo tvrđenje se naziva **baza indukcije**, a drugo **induktivni korak**. Problem se može rešiti rekurzivno ili iterativno. **Rekurzivni pristup** podrazumeva da funkcija sama sebe poziva kako bi rešila problem manje dimenzije (osim u slučaju elementarnih problema) i onda koristi to rešenje za rešavanje početnog. **Iterativni pristup** podrazumeva da promenljive unutar petlje ažuriraju svoje vrednosti u svakom koraku, počevši od vrednosti koje predstavljaju rešenja elementarnih problema pa do vrednosti koje predstavljaju rešenja početnog problema. Ovaj pristup je sličan matematičkoj indukciji pa se naziva i **induktivni**.

Primer: Izračunati sumu niza brojeva.

Rekurzivno

```
int zbir(int niz[], int n) {
        if(n == 0)
                return 0;
        return zbir(niz, n - 1) + niz[n - 1];
}
```

Iterativno

```
int zbir(int niz[], int n) {
        int zbir = 0;
        for(int i = 0; i < n; i++)
                zbir += niz[i];
        return zbir;
}
```

2. Invarijante petlje – pojam, ilustrovanje kroz primere

Invarijanta petlje predstavlja logičke uslove koji važe pre petlje i nakon svakog izvršavanja naredbi u telu petlje, a nakon izvršavanja cele petlje garantuju korektnost algoritma koji ta petlja implementira. Invarijanta suštinski opisuje značenje svih promenljivih unutar petlje.

```
<inicijalizacija>
// ovde važi <invarijanta>
while(<uslov>) {
        // ovde važe i <invarijanta> i <uslov>
        <naredbe>
        // ovde važi <invarijanta>
}
// ovde važi <invrijanta>, a <uslov> ne važi
```

Da bismo dokazali da je neki uslov invarijanta petlje, dovoljno je dokazati da:

- uslov važi pre prvog ulaska u petlju (odgovara bazi indukcije)
- iz pretpostavke da uslov važi pre nekog izvršavanja tela petlja i da je uslov petlje ispunjen dokažemo da uslov važi i nakon izvršavanja tela petlje (odgovara induktivnom koraku)

Ova dva tvrđenja nam garantuju da će uslov važiti i nakon izvršavanja cele petlje ako se ona ikada zaustavi. Na kraju dokazujemo i treće tvrđenje: iz toga da invarijanta važi nakon završetka petlje i da uslov petlje nije uspunjen sledi korektnost algoritma.

Primer: Određivanje minimalnog elementa niza. Invarijanta je: nakon izvršavanja tela petlje promenljiva m ima vrednost minimuma prvih i + 1 elemenata niza.

```
int minNiza(int niz[], int n) {
        int m = a[0];
        for(int i = 1; i < n; i++)
                m = min(m, a[i]);
        return m;
}
```

- baza: nakon 0 izvršavanja petlje m ima vrednost minimuma prvog elementa niza - važi jer smo inicijalizovali m na taj jedan element.
- korak: invarijanta važi pre tela petlje i važi za i < n, onda će važiti i nakon izvršavanja tela petlje - važi jer znamo da pre izvršavanja m ima vrednost minimuma prvih i elemenata niza, zatim ga uporedimo sa $i+1$ elementom niza pa će nakon izvršavanja m biti minimum prvih $i+1$ elemenata niza.
- kad se petlja završi i će imati vrednost n pa na osnovu invarijante m ima vrednost minimuma prvih n elemenata niza.

3. Složenost - vrste složenosti, asimptotska analiza

Bitan faktor pri konstrukciji algoritama jeste koliko resursa on zahteva za svoje izvršavanje. Najčešće su to vreme koje je potrebno za izvršavanje i memorija koja se zauzima. Dakle razmatraju se **vremenska** i **prostorna**, tj.

memorijska složenost. S obzirom na to da savremeni računari raspolažu sa velikom količinom memorije, češće je vreme taj ograničavajući faktor. Složenost zavisi od ulaznih parametara programa. Na primer, nije isto računati prosečnu ocenu iz nekog predmeta za učenike jednog razreda jedne škole i računati tu istu ocenu za učenike svih škola neke države. Takođe, složenost našeg programa ne treba da zavisi od konkretnih ocena koje učenici imaju. Zbog toga se složenost izražava u **funkciji dimenzije ulaznih parametara**, a ne njihovih vrednosti. Neki algoritmi se neće izvršavati isto čak ni za sve ulaze iste veličine, pa je potrebno naći način za opisivanje efikasnosti algoritma na različitim ulazima iste veličine.

- **Analiza najgoreg slučaja** procenu zasniva na najgorem slučaju (slučaj za koji se program najduže izvršava ili zahteva najviše memorije). Ovaj pristup može biti previše pesimističan, ali sa druge strane nam daje garancije da ako je program dovoljno efikasan u najgorem slučaju, biće dovoljno efikasan i u ostalim slučajevima.
- **Analiza prosečnog slučaja** procenu zasniva na proseku izvršavanja većeg broja slučajeva. Da bi ova analiza bila efikasna potrebno je dobro poznavati prostor dopuštenih ulaznih vrednosti, ali i verovatnoću da se svaka od tih vrednosti pojavi na ulazu.
- **Analiza najboljeg slučaja** procenu zasniva na najboljem slučaju i ona nikada nema smisla.
- **Amortizovana analiza** procenu zasniva na izvršavanju određenog broja srodnih operacija i u tim situacijama nije bitno vreme izvršavanja pojedinačnih operacija već samo zbirno vreme izvršavanja svih operacija.

Testiranje rada programa ne mora uvek biti pravi pokazatelj složenost, a možemo dobiti i različite rezultate ako testiranje radimo na različitim operativnim sistemima jer oni mogu biti brži ili sporiji jedni u odnosu na druge. Bolji pristup bi bio da konstruišemo funkciju **f(n)** koja određuje zavisnost broja instrukcija koje algoritam izvršava u odnosu na veličinu ulaza **n**. Algoritmi čija je složenost odozgo ograničena polinomijalnim funkcijama se u principu smatraju efikasnim. Algoritmi čija je složenost odozgo ograničena eksponencijalnom ili faktorijelskom funkcijom se smatraju neefikasnim. Ovakva **asimptotska analiza** daje dobru osnovu za poređenje efikasnosti različitih algoritama. U stvarnosti, vreme izvršavanja programa zavisi i od prevodioca, koji može da optimizuje izvorni kod, ali i od samog operativnog sistema. Međutim, sve optimizacije i razlike u brzini izvršavanja ne mogu da utiču na odnos vremena izvršavanja algoritama različitih klasa složenosti za velike ulaze.

U prevodu, program **kvadratne vremenske složenosti** - f(n) = n^2 će za ogromne ulaze uvek biti manje efikasan u odnosu na program **linearne vremenske složenosti** - f(n) = n. Dodatno, možemo zanemariti i sve koeficijente ispred vodećeg stepena, kao i sve monome manjeg stepena jer oni ne mogu znatno da utiču na efikasnost kod velikih ulaza. Na primer, funkcija složenosti f(n) = $3n^2$ + 6n + 5 je isto što i f(n) = n^2. Gornja granica složenosti se izražava korišćenjem **O-notacije**.

- Ako postoje pozitivna realna konstanta c i prirodan broj n_0 tako da za svaki prirodan broj n veći od n_0 i funkcije f i g važi f(n) $\leq$ c · g(n) onda pišemo f(n) = O(g(n)) i čitamo "f je veliko 'o' od g".

Može se koristiti i **Θ-notacija** koja ne daje samo gornju granicu, već precizno opisuje asimptotsko ponašanje.

- Ako postoje pozitivne realne konstante c_1 i c_2 i prirodan broj n_0 tako da za svaki prirodan broj n veći od n_0 i funkcije f i g važi c_1 · g(n) $\leq$ f(n) $\leq$ c_2 · g(n) onda pišemo f(n) = Θ(g(n)) i čitamo "f je veliko 'teta' od g".

4. Matematičke osnove izračunavanja složenosti - osnovne formule

Da bismo mogli da analiziramo složenost programa potrebno je da vladamo određenim matematičkim aparatom. **Osnovne matematičke formule**:

- **Gausova formula**: 1 + 2 + ... + n = $\frac{n(n+1)}{2}$
- i**-ti član i zbir elemenata aritmetičkog niza**: a_i = a_0 + id, $\sum_{i=0}^{n} a_i$ = $\frac{n}{2}$(2a_0 + nd) = $\frac{n}{2}$(a_0 + a_n)

- **i-ti član i zbir elemenata geometrijskog niza**: $a_i = a_0 \cdot q^i$, $\sum_{i=0}^{n} a_i = a_0 \frac{1-q^{n+1}}{1-q}$
- **Zbirovi stepena**: $1^2 + 2^2 + ... + n^2 = \sum_{k=1}^{n} k^2 = \frac{n(n+\frac{1}{2})(n+1)}{3}$, $1^3 + 2^3 + ... + n^3 = \sum_{k=1}^{n} k^3 = (\frac{n(n+1)}{2})^2$
- **Broj načina da se iz skupa od n različitih izabere k brojeva**: $\binom{n}{k} = \frac{n(n-1)...(n-k+1)}{k!}$

5. Klase složenosti

- **Konstantna složenost O(1)** - algoritmi koji se izvršavaju praktično momentalno, npr. računanje matematičke formule.
- **Logaritamska složenost O($\log n$)** - izuzetno efikasni algoritmi, npr. binarna pretraga.
- **Korenska složenost O($\sqrt{n}$)** - "logaritamski algoritam za one sa jeftinijim ulaznicama", npr. ispitivanje da li je broj prost, faktorizacija broja na proste činioce.
- **Linearna složenost O(n)** - optimalna složenost, kada je za rešenje potrebno gledanje celog ulaza, npr. traženje min/max serije elemenata.
- **Kvazilinearna složenost O($n\log n$)** - "linearni algoritam za one sa jeftinijim ulaznicama", mnogo je bliža linearnoj nego kvadratnoj složenosti, npr. efikasno sortiranje, korišćenje struktura podataka sa logaritamskim vremenom pristupa.
- **Kvadratna složenost O(n^2)** - obično ugnježdene petlje, npr. sortiranje selekcijom, sortiranje umetanjem.
- **Kubna složenost O(n^3)** - obično višestruko ugnježdene petlje, npr. množenje matrica.
- **Eksponencijalna složenost O(2^n)** - izuzetno neefikasno, npr. ispitivanje svih podskupova.
- **Faktorijelna složenost O(n!)** - izuzetno neefikasno, npr. ispitivanje svih permutacija.

Dimenzija ulaza koja se može obraditi za 1s, ako se jedna operacija izvršava za 1ns:

n	$n\log n$	n^2	n^3	2^n	n!
10^9	$40 \cdot 10^6$	32000	1000	30	12

6. Složenost nekih čestih oblika petlji

```
for(int i = 0; i < n; i++)
        // kod složenosti O(1)
```

Složenost petlje je O(n).

```
for(int i = m; i < n; i++)
        // kod složenosti O(1)
```

Složenost petlje je O(n - m).

```
for(int i = 0; i < n; i += 2)
        // kod složensti O(1)
```

Složenost petlje je O(n) jer se ona izvršava $\frac{n}{2}$ puta pa se konstantni faktor $\frac{1}{2}$ može zanemariti.

```
for(int i = 0, j = n - 1; i < j; i++, j--)
        // kod složenosti O(1)
```

Složenost petlje je O(n) jer se pokazivači susreću približno na sredini opsega pa se petlja izvršava oko $\frac{n}{2}$ puta.

```
for(int i = 0; i < n; i++)
        for(int j = 0; j < m; j++)
                // kod složenosti O(1)
```

Složenost petlje je O(mn) jer se spoljašnja petlja izvršava n puta u okviru koje se svaki put unutrašnja petlja izvrši m puta.

```
for(int i = 0; i < n; i++)
        for(int j = i + 1; j < n; j++)
                // kod složenosti O(1)
```

Složenost petlje je $O(n^2)$ jer se telo unutrašnje petlje izvršava (n - 1) + (n - 2) + ... + 2 + 1 puta što je jednako $\frac{n(n-1)}{2}$, tj. $\frac{1}{2}n^2$ - $\frac{1}{2}$n pa se konstantni faktor $\frac{1}{2}$ i monom manjeg stepena mogu zanemariti.

```
for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
                for(int k = 0; k < n; k++)
                        // kod složenosti O(1)
```

Složenost petlje je $O(n^3)$.

```
for(int i = 0; i < n; i++)
        for(int j = i + 1; j < n; j++)
                for(int k = j + 1; k < n; k++)
                        // kod složenosti O(1)
```

Složenost petlje je $O(n^3)$ jer svakom izvršavanju tela petlje odgovara jedna tročlana kombinacija elemenata skupa 0, ..., n - 1. Takvih kombinacija ima $\binom{n}{3} = \frac{n(n-1)(n-2)}{3!}$ pa se konstantni faktor $\frac{1}{6}$ kao i monomi manjeg stepena mogu zanemariti.

```
for(int i = 0; i < m; i++)
        // kod složenosti O(1)
for(int i = 0; i < n; i++)
        // kod složenosti O(1)
```

Složenost petlji je O(m + n) jer se telo prve petlje izvrši m, a druge n puta.

```
for(int i = 0; i < n; i++)
        // kod složenosti O(1)
for(int i = 0; i < n; i++)
        // kod složenosti O(1)
```

Složenost petlji je O(n) jer se telo izvrši n + n = 2n puta pa se konstantni faktor 2 može zanemariti.

```
for(int i = 0; i < n; i++)
        for(int j = i + 1; j < n; j++)
                // kod složenosti O(1)
for(int i = 0; i < n; i++)
        // kod složenosti O(1)
```

Složenost petlji je O(n^2) jer se telo prve petlje izvršava $\frac{n(n-1)}{2}$ puta, a druge n puta što je zanemarivo malo.

```
for(int i = 1; i * i <= n; i++)
        // kod složenosti O(1)
```

Složenost petlje je O($\sqrt{n}$) jer se petlja izvršava dok je $i^2 \leq$ n, tj. $i \leq \sqrt{n}$.

```
for(int i = 1; i < n; i *= 2)
        // kod složenosti O(1)
```

Složenost petlje je O($\log n$) jer se vrednost i duplira u svakom koraku.

```
for(int i = 0; i < 10; i++)
        // kod složenosti O(1)
```

Složenost petlje je O(1) jer je broj izvršavanja konstantan i ne zavisi od veličine ulaza.

```
for(int i = 1; i <= n; i++)
        for(int j = 1; j < i; j *= 2)
                // kod složenosti O(1)
```

Složenost petlje je O(n$\log n$) jer se telo izvršava $\log 1$ + $\log 2$ + ... + $\log n$ puta, što je manje ili jednako $\log n$ + $\log n$ + ... + $\log n$ što je jednako n$\log n$.

```
for(int i = n; i >= 1; i /= 2)
        for(int j = 1; j < i; j++)
                // kod složenosti O(1)
```

Složenost petlje je O(n) jer je broj izvršavanja tela jednak n + $\frac{n}{2}$ + $\frac{n}{4}$ + ... što je jednako n · (1 + $\frac{1}{2}$ + $\frac{1}{4}$ + ...) što je ograničeno sa n · 2 pa se konstantni faktor 2 može zanemariti. **Broj petlji ne odgovara uvek stepenu polinoma!**

```
for(int i = 0; i < n; i++) {
        for(int j = i + 1; j < n && P(a[j]); j++)
                // kod složenosti O(1)
        // kod složenosti O(1)
        i = j;
}
```

Složenost petlje je O(n). Broj izvršavanja unutrašnje petlje zavisi od stanja niza, pa ne možemo unapred znati taj broj, ali možemo izvršiti amortizovanu analizu i izračunati ukupan broj izvršavanja unutrašnje petlje. Unutrašnja petlja uvek kreće od tekuće vrednosti brojača spoljašnje petlje, a spoljašnja petlja nastavlja tamo gde je unutrašnja stala. To znači da će se telo petlje izvršiti najviše n puta. Ovakav kod se može sresti u algoritmima koji analiziraju sve serije uzastopnih elemenata niza koji zadovoljavaju neko svojstvo.

```
int j = 0;
for(int i = 0; i < n; i++) {
        while(j < n && P[j]) {
                // kod složenosti O(1)
                j++;
        }
        // kod složenosti O(1)
}
```

Složenost petlje je O(n) jer unutrašnja petlja ne inicijalizuje j na 0, ali se sve vreme uvećava pa je broj izvršavanja ograničen sa 2n.

```
int l = 0, d = n - 1;
while(l < d) {
        do l++; while(l < d && P(a[l]));
        do d--; while(l < d && Q(a[d]));
        if(l < d)
                // kod složenosti O(1)
}
```

Složenost petlje je O(n). Korišćenjem amortizovane analize, vidimo da se l samo povećava počevši od početka, a d samo smanjuje počevši od kraja niza, sve dok se ne sretnu. To će se desiti u najviše n koraka. Sličan kod se sreće u algoritmu particionisanja niza.

7. Rekurentne jednačine za analizu složenosti, master teorema

Složenost rekurzivnih funkcija se može opisati **rekurentnim jednačinama**. Rekurentne jednačine gde se problem svodi na problem dimenzije za jedan manje od početne:

- T(n) = T(n - 1) + O(1), T(0) = O(1) - rešenje: O(n), npr. traženje minimuma niza
- T(n) = T(n - 1) + O($\log n$), T(0) = O(1) - rešenje: O(n$\log n$), npr. formiranje balansiranog binarnog drveta
- T(n) = T(n - 1) + O(n), T(0) = O(1) - rešenje: O(n^2), npr. sortiranje selekcijom

Rekurentne jednačine gde se problem svodi na dva ili više problema dimenzije za jedan manje od početne:

- T(n) = 2T(n - 1) + O(1), T(0) = O(1) - rešenje O(2^n), npr. Hanojske kule
- T(n) = T(n - 1) + T(n - 2) + O(1), T(0) = O(1) - rešenje: O(2^n), npr. Fibonačijevi brojevi

Rekurentne jednačine gde se problem svodi na jedan ili više problema značajno (bar duplo) manje dimenzije od početne:

- T(n) = T($\frac{n}{2}$) + O(1), T(0) = O(1) - rešenje: O($\log n$), npr. binarna pretraga sortiranog niza
- T(n) = T($\frac{n}{2}$) +O(n), T(0) = O(1) - rešenje: O(n), npr. pronalaženje medijane
- T(n) = 2T($\frac{n}{2}$) + O(1), T(0) = O(1) - rešenje: O(n), npr. obilazak potpunog binarnog drveta
- T(n) = 2T($\frac{n}{2}$) + O(n), T(0) = O(1) - rešenje: O(n$\log n$), npr. sortiranje objedinjavanjem

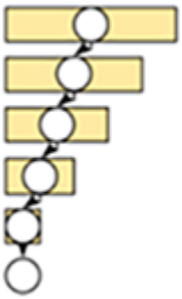
Da bismo stekli intuiciju kako se dobijaju ova asimptotska rešenja, možemo primeniti **"odmotavanje rekurzije"**.

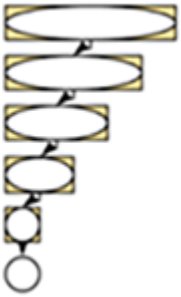
Primer: T(n) = T(n - 1) + O(1), T(0) = O(1)

- T(n) = T(n - 1) + O(1) = T(n - 2) + O(1) + O(1) = ... = T(0) + n · O(1) = O(1) + n · O(1) = O(n)

Primer: T(n) = T(n - 1) + O(n), T(0) = O(1)

- T(n) = T(n - 1) + cn = T(n - 2) + c(n - 1) + cn = ... = T(0) + c · (1 + ... + n) = O(1) + c$\frac{n(n+1)}{2}$ = O(n^2)





Jednačine zasnovane na dekompoziciji problema na manje potprobleme koje su oblika T(n) = aT($\frac{n}{b}$) + O(n^k), T(0) = O(1) se rešavaju na osnovu **master teoreme**:

- Rešenje rekurentne jednačine T(n) = aT($\frac{n}{b}$) + cn^k, T(0) = O(1), gde su a i b celobrojne konstante $\geq$ 1 i c i k pozitivne realne konstante je:

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

Primeri:

- T(n) = 2T($\frac{n}{2}$) + O(1), T(0) = O(1) $\Rightarrow$ a = 2, b = 2, k = 0 $\Rightarrow$ $\log_2 2$ = 1 > k $\Rightarrow$ Θ(n)
- T(n) = 2T($\frac{n}{2}$) + O(n), T(0) = O(1) $\Rightarrow$ a = 2, b = 2, k = 1 $\Rightarrow$ $\log_2 2$ = 1 = k $\Rightarrow$ Θ(n$\log n$)
- T(n) = T($\frac{n}{2}$) + O(n), T(0) = O(1) $\Rightarrow$ a = 1, b = 2, k = 1 $\Rightarrow$ $\log_2 1$ = 0 < k $\Rightarrow$ Θ(n)

8. Odsecanje u linearnoj pretrazi - ilustrovanje kroz primere

9. Odsecanje u linearnoj pretrazi - primeri: prosti brojevi, Eratostenovo sito

Jedan od osnovnih principa za dobijanje efikasnijih algoritama je da računar ne treba da izračunava stvari za koje može unapred proceniti da nisu potrebne za dobijanje konačnog rešenja. Jedan primer se javlja kod linearne pretrage. Kada preskočimo proveru elemenata za koje unapred znamo da ne mogu da zadovolje uslov pretrage kažemo da smo izvršili **odsecanje u linearnoj pretrazi**. Prilikom odsecanja moramo biti posebno pažljivi kako bismo osigurali korektnost algoritma. Trebamo biti sigurni da u delu prostora pretrage koji smo odsekli zaista nije rešenje problema. Neki od najvažnijih primera primene odsecanja su binarna pretraga, bektreking, pretraga u dubinu i pretraga u širinu.

Primer - prost broj: Ispitati da li je uneti prirodan broj n prost.

- Ukoliko redom proveravamo deljivost svim brojevima iz intervala (1, n) složenost algoritma će biti O(n).
- Broj n ne može biti deljiv brojevima većim od $\frac{n}{2}$ pa pretragu možemo vršiti samo do $\frac{n}{2}$. Takođe, ako broj n nije deljiv sa 2 onda nije ni sa jednim parnim brojem, što daje dodatno odsecanje jer možemo proveravati

samo neparne brojeve. Ova odsecanja su suštinski nebitna, jer ne doprinose značajno efikasnosti, unapređen je samo konstantni faktor.

- Delioci broja se uvek javljaju u paru - ako je broj n deljiv sa d tako da je d $\geq \sqrt{n}$ tada je deljiv i brojem $\frac{n}{d} \leq \sqrt{n}$. Odnosno, ako broj nema delilaca manjih od $\sqrt{n}$ onda nema ni delilaca koji su veći ili jednaki $\sqrt{n}$, pa je dovoljno ispitivati delioce iz intervala [2, $\sqrt{n}$] što će nam dati složenost O($\sqrt{n}$).
- Možemo još dodatno unaprediti konstantni faktor. Ako broj nije deljiv ni sa 2 ni sa 3, onda proveravamo samo delioce oblika 6k - 1 i 6k + 1 jer su svi ostali oblici deljivi ili sa 2 ili sa 3.

Primer - Eratostenovo sito: Ispitati da li su brojevi iz intervala [1, n] prosti.

- U naivnom pristupu bismo redom proveravali umnoške svih brojeva i "precrtavali" ih. Složenost će biti O(n $\log n$) jer u svakom koraku spoljašnje petlje d raste pa imamo sve manje i manje izvršavanja unutrašnje petlje.

```
void Eratosten(int n) {
        vector<bool> prost(n + 1, true);
        prost[1] = false;
        for(int d = 2; d <= n; d++)
                for(int x = 2 * d; x <= n; x += d)
                        prost[x] = false;
}
```

- Prvo odsecanje koje možemo napraviti jeste da se unutrašnja petlja izvršava samo ako je d prost broj, jer ako nije njegovi umnošci su već precrtani tokom precrtavanja umnožaka nekog od njegovih prostih faktora. Drugo odsecanje dobijamo jer ne moramo unutrašnju petlju počinjati od 2d, već od d^2. Manji umnošci su već precrtani jer svi imaju prave faktore manje od d. Sada možemo videti da unutrašnja petlja uopšte neće biti izvršena ako je d^2 veće od n, pa opet vršimo odsecanje u spoljašnjoj petlji. Sa svim ovim odsecanjima dobićemo dosta brži algoritam složenosti O(n$\log \log n$).

```
void Eratosten(int n) {
        vector<bool> prost(n + 1, true);
        prost[1] = false;
        for(int d = 2; d * d <= n; d++)
                if(prost[d])
                        for(int x = d * d; x <= n; x += d)
                                prost[x] = false;
}
```

10. Inkrementalnost - ilustrovanje kroz primere

Jedna od osnovnih tehnika poboljšavanja složenosti algoritama jeste izbegavanje izračunavanja istih stvari više puta. Često je potrebno izračunati neku vrednost za različite vrednosti parametra. Izračunavanje je **inkrementalno** ako se rezultujuća vrednost za narednu vrednost parametra računa korišćenjem već izračunatih vrednosti za prethodnu ili više prethodnih vrednosti parametra.

Primer: Izračunati parcijalne zbirove (zbirove prefiksa) niza.

- Primećujemo da prilikom računanja zbirova prefiksa ne moramo svaki zbir računati od početka, već se zbir može dobiti sabiranjem prethodnog zbira sa tekućim elementom. Odnosno važiće $Z_0 = 0$ i $Z_{k+1} = Z_k + a_k$ gde je Z_k zbir prvih k elemenata. Dobili smo seriju brojeva u kojoj se naredni element izračunava na osnovu

prethodnog - **rekurentna serija**. Svaki naredni član računamo u složenosti O(1) pa se izračunavanje svih prefiksnih zbirova može izvršiti u složenosti O(n), gde je n dužina niza. Računanje svakog prefiksnog zbira ispočetka bi dovelo do složenosti $O(n^2)$.

Inkrementalnost je u tesnoj vezi sa induktivno-rekurzivnom konstrukcijom i leži u osnovi mnogih osnovnih algoritama kao što su računanje minimuma, maksimuma, zbira svih elemenata niza, linearna pretraga itd. U svim pomenutim algoritmima krećemo od početne vrednosti, a zatim narednu vrednost računamo na osnovu prethodne ili nekoliko prethodnih što direktno odgovara induktivnom postupku izračunavanja. Slična tehnika se primenjuje i u sklopu dinamičkog programiranja naviše.

Primer: Izračunati maksimum/minimum niza.

- Na početku vrednost min/max niza inicijalizujemo na prvi element, a potom u petlji u svakom koraku tu vrednost poredimo sa narednim elementom niza. U i-tom koraku koristimo već izračunat min/max prvih i elemenata, poredimo ga sa $i+1$ elementom i dobijamo min/max prvih $i+1$ elemenata.

11. Zamena iteracije formulom - ilustrovanje kroz primere

Jedan od načina na koji možemo popraviti složenost naših programa jeste primena matematičkih formula. Umesto da računar izvršava dugotrajna izračunavanja nekad je moguće doći do formula koje će olakšati račun i popraviti složenost. Formula koju dobijemo onda zamenjuje ceo iterativni postupak. **Matematičke formule**

Primer: Izračunati sumu prvih n prirodnih brojeva.

- Sumu prvih n prirodnih brojeva možemo izračunati iterativno preko petlje u složenosti O(n). Primenom Gausove formule zbir je $\frac{n(n+1)}{2}$, pa ćemo dobiti algoritam konstantne složenosti.

Primer - nedostajući broj: U nizu brojeva od 0 do n jedan broj je izostavljen. Pronaći nedostajući element efikasno, bez pamćenja elemenata niza.

- Korišćenjem Gausove formule znamo zbir svih n brojeva, a možemo izračunati zbir unetih brojeva, pa ćemo nedostajući dobiti kao razliku te dve vrednosti. Složenost ovakvog izračunavanja će biti linearna.

Primer: Izračunati NZD dva broja.

- Možemo primeniti Euklidov algoritam zasnovan na oduzimanju: ako su a i b jednaki tada im je i njihov NZD jednak, a u suprotnom se problem svodi na nalaženje NZD-a njihove razlike i manjeg broja. Ovaj algritam će biti složenosti O($\max(a, b)$).

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

- Ako imamo veliku razliku među brojevima, imaćemo veći broj "seckanja kvadrata" što možemo preskočiti ako primenimo Euklidov algoritam zasnovan na deljenju: NZD(a, 0) = a i NZD(a, b) = NZD(b, a % b). Ne moramo

da proveravamo koji je od brojeva veći, jer znamo da će b uvek biti veće od a % b. Složenost je O($\log(max(a,b))$).

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

12. Prefiksni zbirovi, razlike susednih elemenata - ilustrovanje kroz primere

Prefiksne zbirove smo računali pomoću inkrementacije i možemo ih koristiti za računanje zbira elemenata na nekom intervalu. Zbir elemenata na intervalu [a, b] možemo dobiti kao razliku zbira svih elemenata na intervalima [0, b] i
[0, a - 1]. Ova osobina sabiranja može dati efikasnije algoritme jer ako znamo zbirove svih prefiksa niza, što računamo u linearnoj složenosti, onda u konstantnoj složenosti možemo izračunati zbir na nekom segmentu. Dakle, od datog niza možemo dobiti niz zbirova prefiksa u linearnoj složenosti ali važi i obrnuto - od niza prefiksa možemo dobiti originalni niz u linearnoj složenosti. Važi i jače tvrđenje - svaki element niza možemo dobiti u konstantnoj složenosti oduzimanjem dva susedna zbira prefiksa.

Primer: Dato je k upita od kojih svaki traži zbir elemenata na nekom segmentu [a, b].

- S obzirom da imamo k upita, nije efikasno svaki segment računati od početka jer u najgorem slučaju dobijamo složenost O(n · k), gde je n dužina niza. Umesto toga, prvo ćemo izračunati prefiksne sume i onda koristiti pomenuto svojstvo. Niz prefiksnih suma će imati za jedan više elemenata od početnog niza, jer čuvamo i zbir nula elemenata niza. Složenost će sada biti O(n + k) jer nam za računanje prefiksnih zbirova treba O(n) koraka, a za upite O(k) koraka.

```
void upit(int k, vector<int>& a) {
        int n = a.size();
        vector<int> ps(n + 1);
        ps[0] = 0;
        for(int i = 0; i < n; i++)
                ps[i + 1] = ps[i] + a[i];
        while(k--) {
                int a, b;
                cin >> a; cin >> b;
                cout << ps[b + 1] - ps[a];
        }
}
```

Dualan pristup zbirovima prefiska je promena reprezentacije u kojoj umesto niza čuvamo **razlike susednih elemenata**. Povratak na originalni niz dobijamo u linearnoj složenosti računanjem zbirova prefiska niza razlika.

Primer: Dato je k upita od kojih svaki traži da se elementi na nekom segmentu [a, b] uvećaju za datu vrednost.

- Prvo ćemo sa niza preći na niz razlika susednih elemenata. Ako je potrebno promeniti elemente na segmentu

[a, b], onda će se promeniti samo razlike niz[a] - niz[a - 1] i niz[b + 1] - niz[b]. Umesto menjanja svih elemenata u svakom upitu, što bi dovelo do složenosti O(n · k), u svakom upitu ćemo ažurirati samo 2 vrednosti, pa će ukupna složenost biti O(n + k), jer smo niz razlika susednih elemenata formirali u linearnoj složenosti.

13. Primene sortiranja - ilustrovanje kroz primere

Sortiranje podrazumeva ređanje elemenata u odnosu na neki poredak. Sortiranje je jedan od najznačajnijih problema u računarstvu i tokom godina je razvijen veliki broj algoritama za sortiranje kao što su **QuickSort**, **MergeSort**, **HeapSort**, **SelectionSort**, **InsertionSort**, **BubbleSort** i **ShellSort**. Neki od ovih algoritama su jednostavniji ali sporiji, a drugi su komplikovaniji za razumevanje ali efikasniji. Biblioteke savremenih programskih jezika nude gotove implementacije funkcija za sortiranje i uvek je poželjnije koristiti ih umesto ručne implementacije. Složenost sortiranja kod bibliotečkih funkcija je **kvazilinearna - O(n**$\log n$**)**. Biblioteke takođe dopuštaju i navođenje različitih kriterijuma sortiranja zadavanjem funkcija poređenja. Sortiranje ima veliki broj primena i često se koristi kao tehnika pretprocesiranja jer omogućava da se snizi složenost rešenja problema. Sortiranje je korisno jer **grupiše bliske vrednosti**, pa ih možemo efikasno obrađivati. Primenjuje se i prilikom obrade duplikata, poboljšavanja algoritama pretrage i slično.

Primer: Iz niza brojeva efikasno izbaciti sve duplikate.

- Nakon učitavanja i sortiranja niza prolazimo kroz sve elemente počevši od drugog (prvi element niza ne može biti duplikat) i dodajemo ih u rezultujući niz ako nisu isti kao njihov prethodnik. Na kraju možemo resize-ovati niz na onoliko elemenata koliko je ostalo.

14. Oblici binarne pretrage - ilustrovanje kroz primere

Ukoliko imamo sortirani niz vrednosti, **binarna pretraga** nam omogućava nalaženje elemenata u složenosti O($\log n$) što je značajno bolje od linearne pretrage i složenosti O(n). Za binarnu pretragu možemo koristiti ugrađene funkcije:

- **binary_search(begin(a), end(a), x)** - proverava da li je element x unutar sortiranog niza a.
- **equal_range(begin(a), end(a), x)** - vraća par iteratora koji ograničavaju raspon elemenata koji su jednaki x u sortiranom nizu a.
- **lower_bound(begin(a), end(a), x)** - vraća prvi od ta dva iteratora.
- **upper_bound(begin(a), end(a), x)** - vraća drugi od ta dva iteratora.

Traženje elementa u nizu: Algoritam se zasniva na polovljenju intervala prilikom pretrage. Traženi element se na početku poredi sa središnjim i ako je manji znamo da je manji i od svih elemenata desno od središnjeg (jer je niz sortiran) pa taj deo možemo odbaciti, tj. primenjujemo tehniku odsecanja. Analogno, ako je traženi veći od središnjeg odbacujemo levu polovinu niza. Ako nije ni manji, ni veći onda je traženi element jednak središnjem i tu je kraj pretrage - pronašli smo element. Za pretragu možemo koristiti ugrađenu funkciju ili je sami ručno implementirati:

```
bool binarnaPretraga(vector<int>& a, int n, int x) {
        int l = 0, d = n - 1;
        while(l <= d) {
                int s = l + (d - l) / 2;
                if(x < a[s])
                        d = s - 1;
                else if(x > a[s])
                        l = s + 1;
                else
                        return true;
        }
        return false;
}
```

Traženje prelomne tačke: Ukoliko imamo niz čiji elementi su sortirani po tome da li zadovoljavaju neko svojstvo P, možemo razmatrati pronalaženje prelomne tačke niza. Ako u nizu imamo prvo elemente koji ispunjavaju svojstvo P, a zatim one koji ne onda prelomnom tačkom možemo smatrati poziciju poslednjeg elementa koji zadovoljava P ili poziciju prvog elementa koji ne zadovoljava P. Prelomnu tačku možemo lako pronaći korišćenjem funkcija lower_bound i upper_bound ili ručnom implementacijom binarne pretrage. Održavaćemo pozicije l i d i invarijantu da svi elementi levo od l zadovoljavaju P, a da svi desno od d ne zadovoljavaju P. Za elemente na pozicijama $[l, d]$ smatramo da još ne znamo kojoj grupi pripadaju. Slično kao i u klasičnoj binarnoj pretrazi uzimamo element na središnjoj poziciji između l i d i proveravamo da li on ispunjava uslov P. Ako da, onda i svi elementi pre njega zadovoljaju P, pa granicu l pomeramo na $s + 1$.
Ako ne, onda ni ostali elementi posle njega ne zadovoljaju P pa granicu d pomeramo na $s - 1$. Pretragu vršimo sve dok imamo nepoznatih elemenata, tj. dok je $l \leq d$ i kada završimo poslednji element koji zadovoljava P biće na poziciji d, a prvi koji ne zadovoljava P biće na poziciji l. Pronalaženje prvog elementa koji je veći ili jednak traženom:

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

Pronalaženje optimalne vrednosti: Ideja pretrage je da se problem optimizacije "pronaći najmanju vrednost koja zadovoljava uslov" svede na problem odlučivanja "da li data vrednost zadovoljava uslov". Binarnu pretragu možemo primeniti ako problem zadovoljava svojstvo monotonosti. Ovaj oblik pretrage se naziva i **binarna pretraga po rešenju**.

- **Primer**: Drvoseča treba da naseče određenu količinu drveta testerom koja može da se podesi na određenu visinu. Odrediti najveću visinu testere tako da drvoseča iseče dovoljnu količinu drveta. Primetimo da važi svojstvo monotonosti: ako je $v_1 < v_2$ i ako je za visinu v_2 nasečeno dovoljno drva tada će biti nasečeno dovoljno i za visinu v_1. Dakle moguće je primeniti binarnu pretragu po rešenju. Kao granice uzećemo visine 0 za koju je sigurno nasečeno dovoljno drveta i h_{max} koja je visina najvišeg drveta za koju sigurno nije nasečeno dovoljno drveta. Za fiksirane vrednosti visina u jednom prolasku možemo računati da li je isečeno dovoljno drveta dodavanjem razlike visine drveta i visine testere ako je ona pozitivna, tj. ako je drvo veće od visine testere, a inače 0. Ukupna složenost će onda biti otprilike $O(n \log h_{max})$.

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

15. Tehnika dva pokazivača - ilustrovanje kroz primere

Tehnika dva pokazivača obuhvata klasu efikasnih algoritama koje karakteriše postojanje dve ili više brojačkih promenljivih koje se kreću kroz elemente najčešće sortiranog niza. Ono što ih razlikuje od ugnježdenih petlji jeste to što se promenljive uvek kreću u istom smeru, tj. vrednost im se ili stalno povećava ili stalno smanjuje. Realizacija može biti preko jedne petlje koja kontroliše vrednosti obe promenljive ili preko ugnježdenih petlji ali tako da se nakon završetka unutrašnje petlje spoljašnja uvećava do mesta gde se unutrašnja završila. Pošto se svaka promenljiva može promeniti najviše n puta gde je n dužina niza ukupno imamo najviše 2n promena pa dobijamo linearnu složenost, za razliku od klasičnih ugnježdenih petlji gde bismo imali kvadratnu složenost.

- **Primer**: Objediniti elemente dva sortirana niza u jedan niz. Održavaćemo dva brojača koja će prolaziti kroz početne nizove i brojač za rezultujući niz. U svakom koraku dodavaćemo manji od dva trenutna elementa nizova sve dok ne stignemo do kraja jednog od njih. Kada se to desi, sve elemente dužeg niza ćemo prebaciti u rezultujući niz. Ne znamo koji će od njih biti duži, ali nije ni potrebno to da znamo. Na kraju ćemo posebnim petljama prekopirati elemente koji su ostali u datim nizovima. Složenost algoritma će biti linearna jer se tačno jednom prolazi kroz svakih od elemenata početnih nizova - O(m + n).

```
vector<int> objedini(vector<int>& a, vector<int>& b) {
	int m = a.size(), n = b.size();
	vector<int> c(m + n);
	int i = 0, j = 0, k = 0;
	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++];
	return c;
}
```

- **Primer**: Odrediti uniju dva skupa predstavljena pomoću sortiranih nizova. Slično kao objedinjavanje, ali ne znamo unapred koliko će unija imati elemenata pa koristimo push_back za dodavanje i obraćamo pažnju da ne dodajemo iste elemente više puta.

```
vector<int> unija(vector<int>& a, vector<int>& b) {
        int m = a.size(), n = b.size();
        vector<int> c; c.reserve(m + n);
        int i = 0, j = 0;
        while(i < m && j < n) {
                if(a[i] < b[j])
                        c.push_back(a[i++]);
                else if(a[i] > b[j])
                        c.push_back(b[j++]);
                else {
                        c.push_back(a[i]);
                        i++; j++;
                }
        }
        while(i < m)
                c.push_back(a[i++]);
        while(j < n)
                c.push_back(b[j++]);
        return c;
}
```

- **Primer**: Odrediti presek dva skupa predstavljena pomoću sortiranih nizova. Analogno kao unija, ali dodajemo samo elemente koji su jednaki, a inače uvećavamo pokazivač niza u kome je trenutno manji element. Na kraju ne dodajemo elemente koji su višak jer oni nikako ne mogu biti istovremeno u oba niza, tj. ne mogu biti deo preseka.

16. Strukture podataka - klasifikacija, primeri upotrebe

Efikasna organizacija podataka u memoriji predstavlja važan preduslov pisanja efikasnih programa. Pored primitivnih tipova podataka savremeni programski jezici nude veliki broj bibliotečkih struktura podataka. **Strukture podataka** možemo posmatrati preko njihovih operacija, tj. funkcionalnosti koje pružaju pa je bitno poznavati njihov apstraktni interfejs, ali imati i osećaj o složenosti operacija. Po načinu implementacije strukture podataka se mogu grupisati na:

- **sekvencijalne kontejnere** → **array, string, vector, list, forward_list, deque**
- **adaptore kontejnera** → **stack, queue, priority_queue**
- **asocijativne kontejnere** → **set, mulitset, map, multimap**
- **neuređene asocijativne kontejnere** → **unordered_set, unordered_multiset, unordered_map, unordered_mulitmap**

Sekvencijalni kontejneri se koriste za skladištenje serija elemenata. Oni predstavljaju određena uopštenja primitivnih statički alociranih nizova. Karakteriše ih mogućnost obilaska svih elemenata redom (nekad i u oba pravca) kao i mogućnost indeksnog pristupa elementima. Sve osim array (koji je samo omotač primitivnog statičkog niza) su dinamički alocirani i dopuštaju umetanje i brisanje elemenata na početak, sredinu ili kraj pri čemu složenost zavisi od konkretnog kontejnera.

Adaptori kontejnera predstavljaju sloj iznad nekog od postojećih sekvencijalnih kontejnera. Podrazumevani sekvencijalni kontejner im se može promeniti prilikom deklarisanja promenljivih.

- **stack** implementira funkcije steka (**LIFO**) gde se elementi mogu dodavati i uklanjati sa jednog kraja.

- **queue** implementira funkcije reda (**FIFO**) gde se elementi mogu dodavati na jedan, a uklanjati sa drugog kraja.
- **priority_queue** implementira funkcije reda sa prioritetom gde se elementi dodaju u proizvoljnom redosledu, a uklanjaju u nerastućem poretku vrednosti.

Asocijativni kontejneri podrazumevaju da se umetanje, pretraga i brisanje elemenata vrši na osnovu vrednosti, a ne na osnovu pozicije na kojoj se nalaze, tj. elementima se pristupa preko ključa, a ne preko indeksa. Osnovni asocijativni kontejneri su **skupovi** i **mape**, tj. **rečnici**. Skupovi odgovaraju matematičkim skupovima i pružaju efikasno dodavanje i izbacivanje elemenata, kao i proveru da li element pripada skupu. Rečnici čuvaju konačna preslikavanja u kojima se ključevima pridružuju vrednosti. Asocijativni kontejneri se dele na **uređene** i **neuređene**. Uređeni kontejneri nude dodatne funkcionalnosti kao što je ispis elemenata u traženom poretku, ali su ponekad sporiji od neuređenih. Svi oni pretpostavljaju da se njihovi elementi (skupovi) odnosno ključevi (mape) mogu porediti relacijskim operatorom. Obično su implementirani pomoću samobalansirajućih uređenih binarnih drveta i osnovne operacije su im logaritamske složenosti. Takođe postoje i **multiskup**, gde je moguće postojanje više istih elemenata u skupu, i **multimapa**, gde jedan ključ može imati više pridruženiih vrednosti.

Neuređeni asocijativni kontejneri su obično implementirani pomoću heš-tabela i karakteriše ih amortizovana konstantna složenost osnovnih operacija. Oni pretpostavljaju da postoji heš-funkcija koja elemente skupa/ključeve mapa slika u celobrojne vrednosti koje određuju njihovu poziciju. Prilikom heširanja moguće su i **kolizije**, tj. da više elemenata ima iste heš-kodove, tj. da se slikaju na istu poziciju. Jedan od načina rešavanja kolizija jeste da se na pozicijama čuva lista svih elemenata koji imaju tu vrednost heš-funkcije.

17. Strukture podataka sa sekvencijalnim pristupom

Niz - array predstavlja samo tanak omotač oko klasičnog statičkog niza čija je dimenzija poznata u trenutku deklaracije. On omogućava da se klasični statički nizovi mogu koristiti na isti način kao i drugi kontejneri, npr. dodela niza nizu, poređenje nizova sa == i slično.

Vektor - vector je implementiran preko dinamičkog niza koji se realocira po potrebi. Ova implementacija omogućava efikasan pristup, iteraciju u oba smera i dodavanje i skidanje elemenata sa kraja. Može biti neefiksan kada čuva velike objekte zbog česte realokacije.

- **push_back** - dodavanje na kraj
- **pop_back** - uklanjanje sa kraja

Niska - string je implementiran slično kao vektor uz dodatne optimizacije i operacije specifične za podatke tekstualnog tipa.

Jednostruko povezana lista - forward_list je implementirana preko jednostruko povezane liste. Omogućava dodavanje na kraj i početak, kao i brisanje sa početka u vremenu O(1), dok brisanje sa kraja zahteva O(n) jer se mora izmeniti pretposlednji čvor. Umetanje i brisanje ostalih elemenata, kao i indeksni pristup zahtevaju vreme O(n).

Dvostruko povezana lista - list je implementirana preko dvostruko povezane liste. Omogućava i dodavanje i brisanje sa kraja i početka u vremenu O(1). Umetanje i brisanje ostalih elemenata je takođe u vremenu O(1). Mana u odnosu na jednostruke liste je to što je potrebno više memorije, kao i to što pojedinačne operacije mogu biti malo sporije jer se ažuriraju dva pokazivača. Prednost je efikasnije brisanje sa kraja i iteracija unazad. Liste se sve manje koriste jer je alokacija čvorova skupa mada prednost u odnosu na dek i nizove je to što mogu da čuvaju i velike podatke jer ne dolazi do realokacije.

Red sa dva kraja - deque dopušta da se elementi dodaju i uzimaju sa bilo kog kraja reda odnosno kombinuje funkcionalnosti i steka i reda. Implementiran je kao specifična struktura podataka koju možemo zamisliti kao vektor u kome se nalaze pokazivači na blokove elemenata fiksne veličine. Zbog toga on nudi efikasno dodavanje i uklanjanje i sa početka i sa kraja ali i iteraciju u oba smera i indeksni pristup. Sve operacije su složenosti O(1).

- **push_front**/**push_back** - dodavanje na početak/kraj
- **front**/**back** - čitanje sa početka/kraja pod pretpostavkom da red nije prazan
- **pop_front**/**pop_back** - uklanjanje sa početka/kraja
- **empty** - provera da li je red prazan
- **size** - broj elemenata u redu

18. Strukture podataka sa asocijativnim pristupom

Skupovi pružaju efikasno dodavanje i izbacivanje elemenata kao i proveru da li se element nalazi u skupu. Podržan je kroz klase **set** i **unordered_set**. U njima nije dozvoljeno ponavljanje elemenata, pa postoje **multiset** i **unordered_multiset** u kojima jeste.

- **set** je implementiran pomoću uređenog binarnog drveta u čijim se čvorovima nalaze elementi skupa gde nije moguće postojanje više čvorova sa istim vrednostima. Drvo je uređeno binarno drvo 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.
- **multiset** je implementiran pomoću uređenog binarnog drveta u čijim se čvorovima nalaze elementi multiskupa sa dozvoljenim ponavljanjem elemenata ili u čijim čvorovima se nalaze elementi i broj njihovih ponavljanja.
- **unordered_set** i **unordered_multiset** su implementirani preko heš-tabela.

Operacije (kod neuređenog skupa su malo efikasnije):

- **insert** - ubacuje element u skup ako već nije u skupu. Složenost kod set je O($\log n$). Kod unordered_set je najgora složenost O(n), a prosečna O(1). Složenost dodavanja m elemenata je O(m$\log m$).
- **erase** - uklanja element iz skupa ako je u skupu. Složenost ista kao kod insert.
- **find** - vraća iterator na element traženog elementa ili end ako se on ne nalazi u skupu. Složenost ista kao kod insert.
- **size** - vraća broj elemenata skupa.
- **lower_bound** i **upper_bound** - ove operacije imaju uređeni skupovi.

Mape predstavljaju kolekcije podataka u kojima se ključevima pridružuju neke vrednosti. Podržane su kroz klase **map** i **unordered_map**. Ključevi sortirane mape mogu biti samo vrednosti koje se mogu porediti relacijskim operatorom, a ključevi nesortirane mape mogu biti samo vrednosti koje se lako mogu pretvoriti u broj. U njima nije dozvoljeno pridružiti više vrednosti istom ključu, pa postoje **multimap** i **unordered_multimap** u kojima jeste.

- **map** je implementirana pomoću uređenog binarnog drveta u čijim se čvorovima nalaze ključevi i pridružene vrednosti gde nije moguće postojanje više vrednosti za isti ključ.
- **multimap** je implementirana pomoću uređenog binarnog drveta u čijim se čvorovima nalaze ključevi i pridružene vrednosti gde je moguće postojanje više vrednosti za isti ključ.
- **unordered_map** i **unordered_multimap** su implementirane preko heš-tabela.

Operacije (kod neuređene mape su malo efikasnije):

- **find** - pretraga preko ključa, vraća iterator na element ili end ako element nije u mapi. Složenost je O($\log n$).

- **insert** - ubacuje par ključ-vrednost u mapu. Složenost je O($\log n$).
- **erase** - uklanja par ključ-vrednost iz mape. Složenost je O($\log n$).

19. Heširanje - primena i implementacija

Heširanje je jedan od osnovnih načina implementacije neuređenih skupova i mapa. Ove strukture podrazumevaju čuvanje slogova koji se mogu pretražiti na osnovu ključa (kod skupova se čuvaju samo ključevi, dok je kod mapa ključevima pridružena i neka vrednost). Heširanje omogućava efikasno dodavanje novih elemenata, efikasnu pretragu na osnovu ključa, kao i brisanje i izmenu elemenata sa datim ključem. Sa druge strane, heširanje ne omogućava ispis elemenata u uređenom poretku niti pronalaženje ključa najbližeg datom.

Heširanje podrazumeva korišćenje **heš-funkcije** koja svaki ključ preslikava u neki ceo broj koji nazivamo **heš-vrednost** ili **heš-kod**. Svi podaci se čuvaju u **heš-tabeli** i heš-vrednost ključa određuje poziciju elementa u tabeli. Heš-tabela se implementira kao niz koji na jednoj poziciji može čuvati jedan ili više elemenata. Ova mesta u tabeli se ponekad nazivaju i **slotovi** ili **kofe** jer je nekad moguće čuvati i više slogova na jednoj poziciji. U prevodu, ako je tabela dimenzije n onda će heš-funkcija slikati vrednosti iz nekog proizvoljno velikog skupa ključeva u mali skup pozicija tabele [0, n). Najjednostavniji slučaj heširanja jeste onaj gde se može uspostaviti bijekcija između skupa ključeva i dopuštenih pozicija u nizu. Na primer, ako svakom malom slovu engleske abecede želimo da dodelimo vrednost u nizu od 26 elementa heš-funkcija može biti h(c) = c - 'a' koja će a slikati u 0, b u 1 itd.

U stvarnosti je češća situacija kada je broj potencijalnih ključeva dosta veći od dimenzije tabele, kada postoje **kolizije** odnosno više ključeva će imati istu heš-vrednost. Tada je bitno rešiti dva problema, a to su odabir heš-funkcije koja će dovesti do što manje kolizija kao i razrešavanje kolizija kada one nastanu. Jedan od načina rešavanja kolizija jeste da se na pozicijama čuva lista svih elemenata koji imaju tu vrednost heš-funkcije. Na broj kolizija utiče i veličina tabele pa se ona često proširuje kada broj kolizija postane preveliki.

20. Stek - operacije, implementacija, ilustrovanje kroz primere

Stek predstavlja kolekciju podataka gde se oni dodaju po **LIFO** (**last-in-first-out**) principu, tj. element se može dodati i skinuti samo sa vrha steka. Operacije su složenosti O(1), podržane kroz klasu **stack**:

- **push** - postavlja element na vrh steka
- **pop** - skida element sa vrha steka ako nije prazan
- **top** - čita element sa vrha steka ako nije prazan
- **empty** - proverava da li je stek prazan
- **size** - vraća broj elemenata na steku
- **emplace** - umesto push ako se na steku čuvaju parovi ili n-torke

Stek obično podrazumeva da se koristi neki oblik niza ili jednostruko povezane liste. Ukoliko je broj elemenata koji se može naći na steku ograničen može se koristiti statički niz, a ako nije onda se koristi dinamički niz ili lista. U C++ stek za implementaciju koristi vector. Na primer, stack<int> podrazumeva stack<int, vector<int>>.

Primer: Učitavati linije do kraja ulaza, a potom ih ispisati u obrnutom redosledu.

```
void obrniLinije() {
        stack<string> linije;
        string linija;
        while(getline(cin, linija))
                linije.push(linija);
        while(!linije.empty()) {
                cout << linije.top() << endl;
                linije.pop();
        }
}
```

21. Red - operacije, implementacija, ilustrovanje kroz primere

Red predstavlja kolekciju podataka gde se oni dodaju po **FIFO** (**first-in-first-out**) principu, tj. element se dodaje na kraj, a skida sa početka reda. Operacije su složenosti O(1), podržane kroz klasu **queue**:

- **push** - postavlja element na kraj reda
- **pop** - skida element sa početka reda ako nije prazan
- **top** - čita element sa početka reda ako nije prazan
- **empty** - proverava da li je red prazan
- **size** - vraća broj elemenata u redu
- **emplace** - umesto push ako se u redu čuvaju parovi ili n-torke

Red obično podrazumeva da se koristi neki oblik niza ili liste. Ukoliko je broj elemenata koji se može naći na steku ograničen može se koristiti statički niz gde se elementi smeštaju u krug, a ako nije onda se koristi dinamički niz ili lista. U C++ stek za implementaciju koristi deque. Na primer, queue<int> podrazumeva queue<int, deque<int>>.

Primer: Ispisati poslednjih k linija standardnog ulaza.

```
void ispisiLinije(int k) {
        queue<string> linije;
        string linija;
        while(getline(cin, linija)) {
                linije.push(linija);
                if(linije.size() > k)
                        linije.pop();
        }
        while(!linije.empty()) {
                cout << linije.front() << endl;
                linije.pop();
        }
}
```

22. Red sa prioritetom - operacije, implementacija, ilustrovanje kroz primere

Red sa prioritetom je vrsta reda u kome elementi imaju pridružen prioritet, dodaju se u red jedan po jedan, a uvek se uklanja element sa najvećim prioritetom. Operacije dodavanja i skidanja elementa su složenosti O($\log n$), a ostale su O(1) i podržane su kroz klasu **priority_queue**:

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

Red sa prioritetom za skladištenje podataka koristi vector. Na primer, priority_queue<int> podrazumeva priority_queue<int, vector<int>, less<int>> gde je less<int> funkcija za poređenje elemenata i daje opadajuću uređenost elemenata. Moguće je imati red sa rastućim poretkom prosleđivanjem funkcije greater<int>, ali nije moguće imati oba istovremeno. U vektoru su smešteni elementi specijalnog drveta koje se naziva **hip**.

Primer: Ispisati najmanjih k brojeva učitanih sa standardnog ulaza.

```
void kNajmanjih(int n, int k) {
    priority_queue<int, vector<int>, less<int>> red;
    while(n--) {
        int x; cin >> x;
        red.push(x);
        if(red.size() > k)
            red.pop();
    }
    while(!red.empty()) {
        cout << red.top() << endl;
        red.pop();
    }
}
```

23. Hip - primena i implementacija

Hip (**heap**) je struktura podataka koja se koristi za implementaciju redova sa prioritetom. On podržava sledeće operacije:

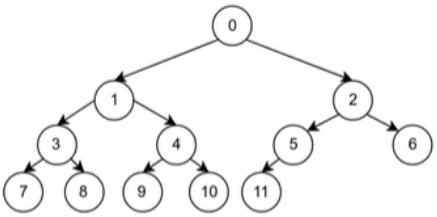
- dodavanje elementa u složenosti O($\log n$)
- uklanjanje najvećeg elementa u složenosti O($\log n$)
- čitanje najvećeg elementa u složenosti O(1)

Umesto hipa koji podržava operacije sa najvećim elementom (**maks-hip**) moguće je razmatrati i hip koji podržava operacije sa najmanjim elementom (**min-hip**), ali nikako oba istovremeno. Elementi hipa se čuvaju u nizu ali zamišljamo da je struktura podataka binarno drvo koje treba da zadovoljava dva uslova:

- **Strukturni uslov** - drvo se popunjava redom, po nivoima i svaki nivo se popunjava redom.

- **Uslov rasporeda vrednosti** - u svakom čvoru drveta nalazi se vrednost veća ili jednaka od vrednosti koje se nalaze u njegovim sinovima.

Ukoliko se čvor nalazi na poziciji k, njegovi sinovi (ako postoje) nalaze se na pozicijama 2k+1 i 2k+2. Roditelj čvora na poziciji m $>$ 0 nalazi se na poziciji $\lfloor \frac{m-1}{2} \rfloor$.



Implementacija operacija:

- **Čitanje najvećeg elementa** - najveći element će se uvek nalaziti u korenu drveta, tj. na početnoj poziciji u nizu. Složenost je O(1).
- **Dodavanje elementa** - element se prvo doda na kraj niza, da bi bio zadovoljen strukturni uslov hipa. Nakon toga vrednost se "penje uz hip", tj. razmenjuje se sa svojim roditeljem sve dok se ne zadovolji uslov rasporeda vrednosti hipa. Složenost je O($\log n$) jer je broj penjanja elementa ograničen visinom drveta, a ona je ograničena logaritmom broja čvorova (jer u potpuno drvo od k nivoa staje 2^k - 1 čvorova).
- **Uklanjanje najvećeg elementa** - iz drveta brišemo poslednji list, a njegovu vrednost upisujemo u koren drveta. Nakon toga vrednost se "spušta niz hip", tj. razmenjuje se sa većim od svojih naslednika sve dok se ne zadovolji uslov rasporeda vrednosti hipa. Složenost je O($\log n$) iz istog razloga kao i za dodavanje elementa.

24. Podeli-pa-vladaj - ilustrovanje kroz primere

25. Podeli-pa-vladaj - brzo sortiranje (QuickSort)

26. Podeli-pa-vladaj - sortiranje objedinjavanjem (MergeSort)

Osnovna ideja tehnike **podeli-pa-vladaj** (**divide-and-conquer**) jeste da je često efikasnije rešavati nekoliko potproblema čija je dimenzija dva ili više puta manja od dimenzije polaznog problema nego rešavati potproblem dimenzije za jedan manje. Kod nekih jednostavnih problema moguće je rešenje dobiti rešavanjem samo jednog takvog potproblema, što dovodi do izrazito efikasnih algoritama. Takođe se naziva i tehnika **razlaganja** ili tehnika **dekompozicije**. Osnovni primeri ove tehnike su sortiranje objedinjavanjem (**MergeSort**) i brzo sortiranje (**QuickSort**), ali i algoritmi obrade binarnog drveta (jer su poddrveta potproblemi koji su u slučaju balansiranih drveta dva puta manji od dimenzije početnog). Kod binarne pretrage se jedan potproblem zanemaruje primenom odsecanja, pa se tehnika u tom slučaju često naziva **smanji-pa-vladaj** (**decrease-and-conquer**).

Ako su potproblemi koji se rešavaju jednake dimenzije mogu se dobiti neke od sledećih rekurentnih jednačina:

- T(n) = 2T($\frac{n}{2}$) + O(1), T(0) = O(1) - rešenje: O(n)
- T(n) = 2T($\frac{n}{2}$) + O($\log n$), T(0) = O(1) - rešenje: O(n)
- T(n) = 2T($\frac{n}{2}$) + O(n), T(0) = O(1) - rešenje: O(n$\log n$)
- T(n) = 2T($\frac{n}{2}$) + O(n$\log n$), T(0) = O(1) - rešenje: O(n$\log^2 n$)

Ako su potproblemi stalno neravnomerne dimenzije moguće je dobiti jednačinu:

- T(n) = T(n - 1) + O(n), T(0) = O(1) - rešenje $O(n^2)$ (složenost najgoreg slučaja algoritma quick sort)

U slučaju smanji-pa-vladaj:

- T(n) = T($\frac{n}{2}$) + O(1), T(0) = O(1) - rešenje: $O(\log n)$
- T(n) = T($\frac{n}{2}$) + O(n), T(0) = O(1) - rešenje: O(n)

QuickSort: U svakom koraku algoritma se jedan element (**pivot**) dovodi na svoje mesto, poželjno blizu sredine niza. Da bismo problem sveli na sortiranje dva manja podniza potrebno je grupisati elemente tako da svi elementi niza manji ili jednaki od pivota budu levo od njega, a svi elementi veći od pivota budu desno od njega, ako niz sortiramo neopadajuće. Ovaj korak se naziva **particionisanje** i vrši se tehnikom dva pokazivača. Nakon toga, rekurzivno se sortiraju leva i desna polovina niza. Izlaz iz rekurzije predstavlja slučaj kada je niz prazan ili jednočlan. Složenost najgoreg slučaja može biti kvadratna ako pivot stalno deli niz na neravnomerne celine. Prosečna složenost je
$O(n\log n)$ i u praksi ovaj algoritam daje dobre rezultate. Primer implementacije QuickSort algoritma:

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

- Ako segment nije neprazan ili jednočlan ($l < d$) za pivot uzimamo proizvoljan element segmenta. Zatim vršimo particionisanje tako da važi da su elementi iz $[l, k]$ manji ili jednaki pivotu, elementi iz (k, i) su veći od pivota, a elementi iz $[i, d]$ su još neispitani. Na kraju razmenimo pivot sa poslednjim manjim ili jednakim elementom i rekurzivno sortiramo levi i desni podniz.

MergeSort: Algoritam deli niz na dva dela čije se dužine razlikuju najviše za 1 (ukoliko niz ima paran broj elemenata onda će podnizovi biti iste dužine). Rekurzivno se sortiraju oba podniza i zatim se sortirane polovine objedinjuju. Objedinjavanje se vrši tehnikom dva pokazivača i potrebno je koristiti pomoćni niz. Izlaz iz rekurzije predstavlja slučaj kada je niz jednočlan (slučaj praznog niza nastupa samo ako je polazni niz prazan). Složenost najgoreg slučaja je $O(n\log n)$ što znači da je MergeSort mnogo brži od SelectionSort ili InsertionSort algoritama čija je složenost kvadratna. Primer implementacije MergeSort algoritma:

```
void merge_sort(vector<int>& a, int l, int d, vector<int>& tmp) {
        if(l < d) {
                int s = l + (d - l) / 2;
                merge_sort(a, l, s, tmp);
                merge_sort(a, s + 1, d, tmp);
                int i = l, m = s, j = s + 1, n = d, k = l;
                while(i <= m && j <= n)
                        tmp[k++] = a[i] <= a[j] ? a[i++] : a[j++];
                while(i <= m)
                        tmp[k++] = a[i++];
                while(j <= n)
                        tmp[k++] = a[j++];
                for(int p = l; p <= d; p++)
                        a[p] = tmp[p];
```

```
        }
    }
```

- Ako segment nije prazan ili jednočlan ($l < d$) uzimamo sredinu segmenta i rekurzivno sortiramo levi i desni podniz. Nakon toga sortirane podnizove spajamo u pomoćni, a onda elemente pomoćnog kopiramo u početni niz.

27. Pretraga sa povratkom - ilustrovanje kroz primere

Algoritmi pretrage sa povratkom (**bektreking**) poboljšavaju tehniku grube sile tako što provere vrše i tokom generisanja kandidata za rešenja i tako što odbacuju parcijalno popunjene kandidate za koje se može unapred utvrditi da se više ne mogu proširiti do ispravnog rešenja. Ovi algoritmi su korisni pri rešavanju **problema zadovoljenja ograničenja** koji podrazumevaju ispitivanje da li postoji kombinatorni objekat koji ispunjava neke uslove, ali i pri rešavanju **problema optimizacije uz ograničenja** koji podrazumevaju pronalaženje kombinatornog objekta koji minimalizuje ili maksimalizuje neku funkciju u grupi datih objekata.

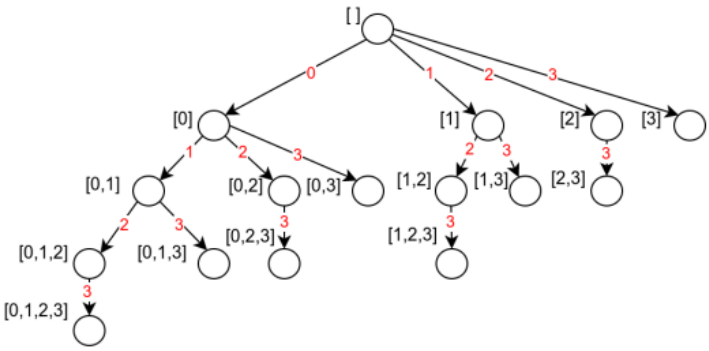
Dakle, umesto da se prvo generišu svi kombinatorni objekti pa da se tek onda proverava da li ispunjavaju neki uslov, bektreking vrši odsecanja ako u nekom trenutku može da ustanovi da data grupa kombinatornih objekata ne može biti rešenje problema. Na primer, ako proveravamo da li postoji podskup nekog skupa brojeva čiji je zbir jednak traženom broju ne moramo grubom silom generisati sve podskupove i proveravati sve zbirove. Ukoliko ustanovimo da je zbir prva dva broja veći od traženog zbira automatski možemo odseći sve podskupove koji sadrže ta dva broja zajedno.

Dodatna optimizacija se može postići kroz **zaključivanje** (**inferrence**). Zaključivanje predstavlja funkciju koja ponekad može da "pogodi" vrednost koja je moguće rešenje problema. Ako je u nekim koracima pretrage zaključivanje previše komplikovano, može i da se preskoči. Algoritam će biti korektan i bez bilo kakvog oblika zaključivanja. Na primer, ako prilikom popunjavanja magičnog kvadrata popunimo sve elemente jedne vrste osim jednog, lako možemo da izračunamo vrednost koja mora biti upisana na preostalom mestu.

Kod optimizacionih problema odsecanje može da nastupi i kada se proceni da u delu drveta koje se trenutno pretražuje ne postoji rešenje koje je bolje od najboljeg trenutno pronađenog rešenja. Drugim rečima, rešenja pronađena u dosadašnjem delu pretrage koriste se da bi se odredile granice na osnovu kojih se vrši odsecanje u drugim delovima pretrage. Ovaj oblik optimizacije naziva se **grananje sa ograničavanjem** (**branch and bound**). Efikasnost algoritama zasnovanih na bektrekingu uveliko zavisi od kvaliteta odsecanja. Složenost najgoreg slučaja obično ostaje eksponencijalna, što je složenost pretrage grubom silom, ali pažljivo odabrani kriterijumi odsecanja mogu odseći jako velike delove pretrage i značajno ubrzati proces pretrage.

28. Generisanje kombinatornih objekata - ilustrovanje kroz primere

U velikom broju slučaja, prostor potencijalnih rešenja nekog problema predstavlja prostor određenih **kombinatornih objekata** kao što su svi podskupovi nekog konačnog skupa, sve varijacije sa ili bez ponavljanja, sve kombinacije sa ili bez ponavljanja, sve permutacije, sve particije i slično. Ovi objekti se obično predstavljaju različitim n-torkama brojeva. Na primer, podskup {a_0, a_2, a_3} skupa od 6 članova se na jedan način može predstaviti n-torkom (0, 2, 3), a na drugi način n-torkom 101100. Takođe, objekti se mogu predstaviti i drvetom koje odgovara procesu njihovog generisanja. Na primer, svi podskupovi četvoročlanog skupa mogu se predstaviti drvetom:



Prilikom generisanja objekata često je poželjnoređati ih određenim redom, najčešće u **leksikografskom poretku**. Torka $a_0 \ldots a_{m-1}$ leksikografski prethodi torci $b_0 \ldots b_{n-1}$ 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, svi podskupovi skupa {1, 2, 3} poređani leksikografski su $\emptyset$, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}.

Primer: Za dati podskup brojeva {1, . . ., n} odrediti podskup koji u leksikografskom poretku sledi neposredno iza njega. Do narednog podskupa možemo doći na dva načina:

- Kada se naredni podskup dobija dodavanjem elementa na prethodni podskup, tj. proširivanjem. Da bi se održao leksikografski poredak, dodati element mora biti najmanji mogući, tj. za jedan veći od poslednjeg elementa trenutnog podskupa (osim ako je u pitanju prazan skup, koji se proširuje sa 1). Jedini slučaj kada proširivanje nije moguće jeste kada je poslednji element trenutnog podskupa najveći mogući. Na primer, za podskupove četvoročlanog skupa podskup 123 se proširuje i sledi mu podskup 1234, dok podskup 234 ne može da se proširi.
- Kada se naredni podskup dobija uklanjanjem elemenata iz trenutnog podskupa i izmenom ostalih, tj. skraćivanjem. Skraćivanje funkcioniše tako što se završni element izbaci, a najveći od preostalih uveća za 1 (on ne može biti najveći jer su elementi unutar podskupa strogo rastući, pa je uvećavanje za 1 uvek moguće). Ako na kraju ostane prazan skup naredni podskup ne postoji. Na primer, za podskupove četvoročlanog skupa podskup 234 se skraćuje i sledi mu podskup 24, dok podskup 4 skraćivanjem daje prazan skup pa je to poslednji podskup i ne postoji naredni.

29. Dinamičko programiranje - pojam, ilustrovanje kroz primere

Prilikom izvršavanja rekurzivne funkcije u praksi često dolazi do **preklapanja rekurzivnih poziva**, tj. identični rekurzivni pozivi se izvršavaju više puta. Ako se to dešava često program postaje neefikasan. Tehnika **dinamičkog programiranja** pruža efikasnije rešenje problema tako što vremensku složenost popravlja angažovanjem dodatne memorije u kojoj se beleže rezultati izvršenih rekurzivnih poziva. Dinamičko programiranje dolazi u dva oblika:

- **Memoizacija** ili **dinamičko programiranje naniže** zadržava rekurzivnu definiciju ali u dodatnoj strukturi podataka (najčešće u nizu ili matrici) beleži rezultate rekurzivnih poziva da bi u narednim pozivima samo očitala vrednosti iz strukture.
- **Dinamičko programiranje naviše** u potpunosti uklanja rekurziju (koja se zamenjuje induktivnom konstrukcijom) i pomoćnu strukturu podataka popunjava iscprno u nekom sistematičnom redosledu.

Kod dinamičkog programiranja naviše jasno je da se funkcija poziva za sve parametre manje od traženog, dok se kod memoizacije može desiti da se funkcija ne poziva za neke parametre. Iz tog razloga možemo pomisliti da je memoizacija efikasnija ali u praksi je uglavnom potrebno računati vrednost rekurzivne funkcije za veliki broj različitih parametara pa se prednost memoizacije retko sreće.

Primer: Fibonačijev niz.

- Klasična rekurzivna implementacija već pri računanju 50. člana Fibonačijevog niza radi veoma sporo jer se funkcija za iste parametre poziva veliki broj puta.

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

- Tehnikom dinamičkog programiranja naniže (memoizacijom) kroz globalni niz čuvamo vrednosti već izračunatih članova niza i na početku svakog poziva proveravamo da li tu vrednost imamo već sačuvanu. Za niz $memo$ treba alocirati n+1 element. Složenost sada više nije eksponencijalna, već linearna.

```
long Fib(int n) {
        if(memo[n] != 0)
                return memo[n];
        if(n == 0 || n == 1)
                return memo[n] = 1;
        return memo[n] = Fib(n - 1, memo) + Fib(n - 2, memo);
}
```

- Tehnikom dinamičkog programiranja naviše potpuno eliminišemo rekurziju i sami popunjavamo pomoćnu strukturu od početka do kraja za sve potrebne vrednosti. Krajnja vrednost će na kraju biti na poziciji n pomoćne strukture.

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

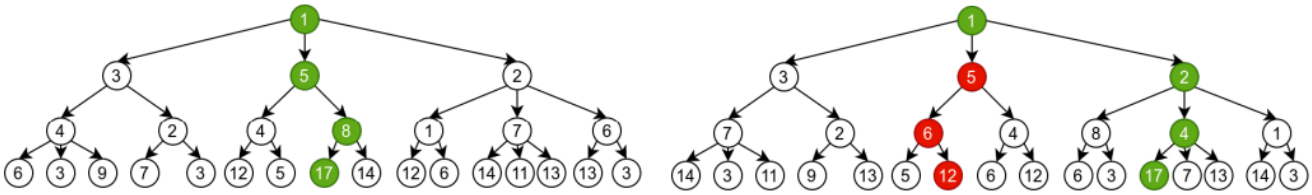
- Kod dinamičkog programiranja naviše možemo izvršiti dodatnu memorijsku optimizaciju jer primećujemo da ne moramo pamtiti sve vrednosti, već samo dva prethodnika.

```
long Fib(int n) {
        long prethodni = 1, pretprethodni = 1;
        for(int i = 2; i <= n; i++) {
                long tekuci = prethodni + pretprethodni;
                pretprethodni = prethodni;
                prethodni = tekuci;
        }
        return prethodni;
}
```

30. Pohlepni algoritmi - pojam, ilustrovanje kroz primere

Algoritmi zasnovani na iscrpnoj pretrazi do rešenja dolaze kroz niz koraka gde u svakom koraku analiziraju sve mogućnosti. Algoritmi kod kojih se umesto analize različitih mogućnosti u svakom koraku uzima neko **lokalno optimalno rešenje** nazivaju se **pohlepni** ili **gramzivi** (**greedy**) **algoritmi**.

- **Primer**: Maksimum drveta pretrage. U prvom slučaju, biranje lokalnog optimalnog rešenja daje dobar rezultat, a u drugom ne.



- **Primer**: Vraćanje kusura sa što manje novčanica. Ako imamo novčanice 1, 2, 5, 10, 20, 50 i 100 i treba da vratimo kusur od 143 dinara, gramziva strategija daje optimalno rešenje - uvek vraćamo najveću novčanicu koja je manja ili jednaka od preostalog iznosa (143 = 100 + 20 + 20 + 2 + 1). Ako imamo novčanice 1, 3 i 4 i treba da vratimo kusur od 6 dinara, gramziva strategija daje rešenje 6 = 4 + 1 + 1, ali ono nije optimalno jer kusur možemo vratiti sa dve novčanice (6 = 3 + 3).

Zaključujemo da ove algoritme ima smisla primenjivati samo kod problema kod kojih postoji garancija da će takvi izbori na kraju dovesti do **globalno optimalnog rešenja**. Nekada se primenjuju i kada nemamo garanciju da će rešenje biti optimalno, ali garantuju da će dovoljno brzo naći dovoljno kvalitetna suboptimalna rešenja. Ovi algoritmi se nazivaju **gramzive heuristike**. S obzirom da gramzivi algoritmi ne vrše ispitivanje različitih slučajeva ili iscrpnu pretragu po pravilu su veoma efikasni. Sa druge strane, odsecanje koje se vrši mora biti opravdano i bitno je dokazati korektnost ovih algoritama. Potrebno je dokazati:

- da gramziva strategija uvek daje korektno rešenje, tj. da zadovoljava uslove zadatka.
- da gramziva strategija uvek daje optimalno rešenje, tj. da ne postoji ni jedno korektno rešenje koje je bolje. Ovaj uslov je teže dokazati i postoje različite tehnike dokazivanja:
  - **tehnika razmene** (**exchange**): bilo koje optimalno rešenje može se transformisati u rešenje koje daje naša strategija. Na primer, ako imamo dva rasporeda časova koji su optimalni, razmenama časova možemo dobiti raspored koji je dala naša strategija.
  - **tehnika "pohlepno rešenje je uvek ispred"** (**greedy stays ahead**): rešenje koje daje naša strategija je u svakom koraku bolje od ostalih rešenja do kojih se može doći u tom trenutku, tj. pohlepno rešenje nikad ne kasni. Na primer, u svakom koraku prilikom dodavanja časova naš raspored će uvek biti bolji od ostalih rasporeda.
  - **tehnika granice** (**structural bound**): odredi se teorijska granica vrednosti optimuma i dokaže da je naše rešenje upravo ta granica koju je moguće dostići teorijski.