

# 4

## Sortiranje

Ovo poglavlje fokusirano je na osnovne tehnike i algoritme za sortiranje. U cilju lakšeg razumijevanja efikasnosti algoritama za sortiranje, prvo su uvedeni i prezentovani osnovni pojmovi analize složenosti algoritama, a zatim su prikazani osnovni pojmovi i klasifikacije tehnika za sortiranje. Nakon toga su prikazani osnovne tehnike i algoritmi za sortiranje kolekcija podataka smještenih u operativnoj memoriji računara.

### 4.1. Analiza složenosti algoritama

#### 4.1.1. Osnovni pojmovi

Analiza složenosti algoritama sprovodi se s ciljem utvrđivanja uticaja broja ulaznih podataka na vrijeme izvršavanja i količinu potrebne memorije za izvršavanje programa. Zavisnost vremena izvršavanja od broja ulaznih podataka naziva se **vremenska složenost**, dok se zavisnost potrebne količine memorije od broja ulaznih podataka naziva se **prostorna složenost**. U ovom poglavlju, fokus je na analizi vremenske složenosti.

Analiza složenosti pomaže u donošenju odluke o izboru odgovarajućeg algoritma za rješavanje konkretnog problema, budući da neki algoritmi imaju praktičnu primjenu samo za mali broj ulaznih podataka.

Složenost nekog algoritma može da se određuje *a priori* i *a posteriori*. Određivanje složenosti *a priori* podrazumijeva procjenu složenosti izvršavanja, bez stvarnog izvršavanja programa i mjerjenja potrebnog vremena i memorije, dok *a posteriori* podrazumijeva eksperimentalno utvrđivanje složenosti.

U praksi je mnogo značajnija procjena složenosti (*a priori*) u odnosu na tačno (*a posteriori*) određivanje složenosti, jer se složenost nekog algoritma može relativno lako procijeniti. Ponekad je složenost takva da bi eksperimentalno mjerjenje bilo dugotrajno ili neizvodljivo.

Složenost algoritma može da se procjenjuje za **najbolji slučaj** (engl. *best case scenario*), **najgori slučaj** (engl. *worst case scenario*) i **prosječan ili tipičan slučaj** (engl. *average/typical scenario*). U najboljem slučaju, izvršavanje programa biće najkraće, a u najgorem slučaju biće najduže. Praktično, za ocjenu složenosti mnogo veći značaj ima analiza najgoreg slučaja, jer izvršavanje programa nikad neće biti duže, bez obzira na to kakva je kolekcija ulaznih podataka. Na primjer, prepostavimo da treba procijeniti složenost nekog algoritma za sortiranje niza u rastućem poretku. U najboljem slučaju, polazni niz je već uređen u rastućem poretku – svi elementi su na odgovarajućim pozicijama i nije potrebno premještanje nijednog elementa. U najgorem slučaju, polazni niz je uređen u obrnutom (opadajućem) poretku i neophodno je premještanje svih elemenata.

Procjena složenosti nekog algoritma zasniva se na **procjeni broja operacija** (algoritamskih koraka) potrebnih za obradu ulaznih podataka, nezavisno od programskog jezika u kojem će algoritam biti implementiran i konkretnog računara na kojem će se program izvršavati. Na primjer, ako je za neku operaciju nad nekim podatkom potreban jedan algoritamski korak, tada je za ispis  $n$  članova nekog niza potrebno  $n$  koraka. Ako jedan uređeni par od dva broja možemo formirati u jednom koraku, onda je za formiranje svih uređenih parova na osnovu  $n$  brojeva, potrebno  $n^2$  koraka. **Stvarno vrijeme izvršavanja programa** biće proporcionalno broju potrebnih koraka i zavisiće od konkretnog hardverskog i softverskog okruženja u kojem će se program izvršavati. Dakle, procijenjeno vrijeme izvršavanja  $T(n)$ , odnosno broj potrebnih koraka, može se predstaviti kao funkcionalna zavisnost broja algoritamskih koraka od broja ulaznih podataka, koja se naziva **funkcija složenosti**, tj.

$$T(n) = f(n).$$

Ako za izvršavanje nekog programa (nad istim brojem podataka) treba više algoritamskih koraka, kažemo da je njegova složenost veća. Dakle, algoritam za formiranje uređenih parova ima veću složenost ( $T(n) = n^2$ ) od algoritma za ispis niza ( $T(n) = n$ ).

Složenost algoritma uglavnom ne igra bitnu ulogu na malom broju ulaznih podataka, ali s porastom broja podataka vrijeme izvršavanja programa može značajno da raste za algoritme sa većom složenošću. Na primjeru zavisnosti iz prethodnog primjera, ako je za ispis 100 brojeva potrebna jedna sekunda, za ispis 1.000 podataka biće potrebno deset sekundi (10 puta više podataka zahtijeva 10 puta više vremena), ali ako je za formiranje svih uređenih parova na osnovu 100 brojeva potrebno 100 sekundi, tada će za formiranje svih uređenih parova na osnovu 1.000 brojeva biti potrebno 10.000 sekundi (10 puta veća ulazna kolekcija zahtijeva 100 puta više vremena).

Primjer 4.1. ilustruje procjenu stvarnog vremena izvršavanja programa, u zavisnosti od broja ulaznih podataka i složenosti algoritma. Ilustrativni podaci pokazuju da polinomska složenost (funkcije oblika  $n^k$ ) zahtijevaju mnogo manje vremena nego eksponencijalna ( $k^n$ ) i faktorijelska ( $n!$ ). **Algoritmi koji imaju polinomsku složenost smatraju se efikasnim**, dok se algoritmi sa eksponencijalnom i faktorijelskom složenošću smatraju neefikasnim.

**Primjer 4.1:**

Ako prepostavimo da se svaka operacija u procesoru izvrši za  $1\text{ ns}$ , tada su u tabeli prikazana procijenjena stvarna vremena izvršavanja u zavisnosti od broja ulaznih podataka i funkcije složenosti algoritma.

$n$	$f(n)$						
	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$	$n!$
10	0,003 $\mu s$	0,01 $\mu s$	0,033 $\mu s$	0,1 $\mu s$	0,1 $\mu s$	1,0 $\mu s$	3,63 ms
100	0,007 $\mu s$	0,1 $\mu s$	0,644 $\mu s$	10 $\mu s$	1 ms	$4 \cdot 10^{13} \text{ god}$	
1.000	0,010 $\mu s$	1,0 $\mu s$	9,966 $\mu s$	1 ms	1 s		
10.000	0,013 $\mu s$	10 $\mu s$	130 $\mu s$	0,1 s	16,7 min		
100.000	0,017 $\mu s$	100 $\mu s$	1,67 ms	10 s	11,57 dan		
1.000.000	0,020 $\mu s$	1 ms	19,93 ms	16,7 min	31,7 god		

Na osnovu funkcije složenosti algoritma može lako da se odredi i broj ulaznih podataka koje program može da obradi u jedinici vremena.

**Primjer 4.2:**

Ako prepostavimo da se svaka operacija u procesoru izvrši za  $1\text{ ns}$ , tada su u tabeli prikazane procijenjene količine podataka koje će biti obrađene u zavisnosti od trajanja obrade i funkcije složenosti algoritma.

$t$	$f(n)$					
	$n$	$n \log n$	$n^2$	$n^3$	$2^n$	$n!$
1 ms	$10^6$	63.000	1.000	100	20	9
1 s	$10^9$	$40 \cdot 10^6$	32.000	1.000	30	12
1 min	$60 \cdot 10^9$	$1,9 \cdot 10^9$	245.000	3.900	36	14

**4.1.2. O-notacija**

Gornja granica funkcije složenosti izražava se korištenjem **O-notacije** (čita se "veliko O").

**Definicija:** Ako postoje pozitivna realna konstanta  $c$  i prirodan broj  $n_0$ , takvi da za funkcije  $f$  i  $g$  važi  $f(n) \leq c \cdot g(n)$  za svaki prirodan broj  $n > n_0$ , onda pišemo  $f(n) = O(g(n))$  i čitamo " $f$  je veliko  $O$  od  $g$ ".

Drugim riječima, za procjenu vremena izvršavanja bitno je odrediti red funkcije složenosti  $f(n)$ , jer za dovoljno veliko  $n$ , red veličine dominantno utiče na procjenu, tj.

$$T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = O(n^k).$$

Prethodna interpretacija ima veliki praktični značaj za procjenu složenosti, jer je dovoljno odrediti samo red funkcije složenosti, bez potrebe da se ona potpuno tačno odredi (vidjeti primjer 4.3). Na primjer, za  $T(n) = n^2 + 3n + 1$  i ulaznu kolekciju sa  $n = 1000$  podataka, dobijamo da ukupan broj operacija iznosi  $1,003,001 \approx 10^6 = n^2$ , dok za kolekciju sa milion podataka dobijamo  $1.000009 \cdot 10^{12} \approx 10^{12} = n^2$ .

**Primjer 4.3:**

Prepostavimo sljedeći C program, koji sa standardnog ulaza učitava prirodan broj  $n$ , a zatim ispisuje prvih  $n$  redova trougla popunjeno zvjezdicama.

```
#include <stdio.h>
int main()
{
    int n;
    printf("n=");
    scanf("%d", &n);
    if (n>0)
    {
        for (int i=1; i<=n; i++)
        {
            for (int j=1; j<=i; j++)
                printf("*");
            printf("\n");
        }
    }
    return 0;
}
```

Za prve četiri naredbe, analiza je jednostavna – svaka naredba izvršiće se jednom, što je ukupno četiri operacije (koraka).

Unutrašnja `for` petlja sadrži: jednu inicijalizaciju,  $i + 1$  provjeru uslova,  $i$  inkrementovanja brojača te  $i$  ispisa zvjezdice. Dakle, unutrašnja petlja sadrži ukupno  $1 + (i + 1) + i + i = 3i + 2$  koraka.

Svaki ciklus spoljašnje petlje sadrži jednu unutrašnju petlju i jedan poziv funkcije `printf` za prelazak u sljedeći red, što znači da svaki ciklus spoljašnje petlje sadrži  $3i + 2 + 1 = 3i + 3$  koraka.

Spoljašnja petlja sadrži: jednu inicijalizaciju,  $n + 1$  provjeru uslova,  $n$  inkrementovanja brojača te sve korake ciklusa. U  $i$ -tom koraku spoljašnje petlje ciklus sadrži  $3i + 3$  koraka, pa je ukupan broj koraka u svim ciklusima spoljašnje petlje

$$T'(n) = \sum_{i=1}^n (3i + 3) = 3 \sum_{i=1}^n (i + 1) = 3 \sum_{i=1}^n i + 3 \sum_{i=1}^n 1 = 3 \frac{n(n + 1)}{2} + 3n.$$

Ukupan broj koraka spoljašnje petlje je

$$T''(n) = 1 + (n + 1) + n + 3 \frac{n(n + 1)}{2} + 3n = \frac{3n^2 + 13n + 4}{2}.$$

Kad ukupnom broju koraka spoljašnje petlje dodamo pet koraka ostalih naredbi u programu, dobijamo funkciju složenosti datog programa

$$T(n) = 4 + \frac{3n^2 + 13n + 4}{2} + 1 = \frac{3n^2 + 13n + 14}{2}.$$

Red funkcije složenosti je 2 i važi  $T(n) = O(n^2)$ .

Kao što je ranije navedeno, za procjenu složenosti nije neophodno da se određuje funkcija složenosti, nego samo njen red. Iz prethodne analize može se uočiti da je spoljašnja petlja diktirala složenost algoritma, čija je složenost reda 2. Njena složenost je reda dva, jer sadrži ugnježđenu petlju. Dakle, s obzirom na to da smo u programu imali ugnježđenu petlju, odmah smo mogli zaključiti da je red funkcije složenosti jednak 2.

---

### Klase složenosti algoritama

Osnovne klase složenosti algoritama prikazane su u tabeli 4.1. Za dovoljno veliko  $n$  važi:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < \dots < O(n^k) < \dots < O(k^n) < O(n!).$$

Algoritmi koji imaju logaritamsku složenost su izuzetno efikasni; algoritmi sa linearom složenošću su optimalni (очекuje se pristup svakom elementu u kolekciji); algoritmi sa polinomskom složenošću smatraju se efikasnim, a algoritmi sa eksponencijalnom i faktorijelskom složenošću smatraju se neefikasnim. Treba uočiti da eksponencijalna funkcija složenosti ne može odozgo da se ograniči nijednom polinomskom funkcijom.

**Tabela 4.1:** Osnovne klase složenosti algoritama

$O(g(n))$	Klasa složenosti
$O(1)$	<b>Konstantna složenost.</b> Složenost ne zavisi od broja ulaznih podataka. Primjer: svaki algoritam bez petlje.
$O(\log n)$	<b>Logaritamska složenost.</b> Složenost zavisi od dimenzije ulazne kolekcije, ali se ne pristupa svim elementima, nego se problem rješava particionisanjem (dijeljenjem) kolekcije do jedinične kolekcije (uz odbacivanje "viška"). Primjer: binarno pretraživanje niza ("odbacivanje polovine").
$O(n)$	<b>Linearna složenost.</b> U najgorem slučaju mora se pristupiti svakom ulaznom podatku, pri čemu obrada svakog podatka ima konstantnu složenost. Primjer: učitavanje niza, pretraživanje nesortiranog niza itd.
$O(n \log n)$	<b>Linearno-logaritamska složenost.</b> Ulazna kolekcija dijeli se na particije (sve do jedinične), a particije se obrađuju sekvencijalno. Primjer: sortiranje spajanjem sortiranih particija ("merge-sort").
$O(n^2)$	<b>Kvadratna složenost.</b> U najgorem slučaju mora da se obradi svaki element ulazne kolekcije, pri čemu obrada uključuje dodatnu sekvencijalnu obradu. Primjer: dvije ugnježđene petlje.
$O(n^k)$ $(k > 2)$	<b>Stepena složenost.</b> Generalizacija algoritama sa kvadratnom složenošću. Primjer: $k$ ugnježđenih petlji (npr. množenje matrica).
$O(k^n)$ $(k > 1)$	<b>Eksponencijalna složenost.</b> Primjer: ispisivanje svih podstringova nekog stringa).
$O(n!)$	<b>Faktorijelska složenost.</b> Primjer: generisanje permutacija.

### 4.1.3. Druge notacije za reprezentaciju složenosti

Pored O-notacije, koja se koristi za procjenu složenosti u najgorem slučaju (ograničenost funkcije složenosti odozgo), u primjeni su i druge notacije.

#### $\Omega$ -notacija

$\Omega$ -notacija koristi se za procjenu složenosti algoritma u najboljem slučaju i služi za reprezentaciju reda veličine donje granice funkcije složenosti.

**Definicija:** Ako postoje pozitivna realna konstanta  $c$  i prirodan broj  $n_0$ , takvi da za funkcije  $f$  i  $g$  važi  $f(n) \geq c \cdot g(n)$  za svaki prirodan broj  $n > n_0$ , onda pišemo  $f(n) = \Omega(g(n))$  i čitamo "f je omega od g".

Za ilustraciju, posmatrajmo algoritam za pretraživanje niza. Pretraživanje je operacija kojom se provjerava da li u nekoj kolekciji postoji odgovarajući podatak. Ako niz nije sortiran, u najgorem slučaju treba da pristupimo svakom elementu niza da bismo provjerili da li u nizu postoji odgovarajući podatak, što znači da je  $T(n) = O(n)$ . U najboljem slučaju, traženi podatak može biti na početku niza i pretraživanje se uspješno završava bez potrebe da se pristupa ostalim elementima niza, što znači da je  $T(n) = \Omega(1)$  – u najboljem slučaju složenost je konstantna.

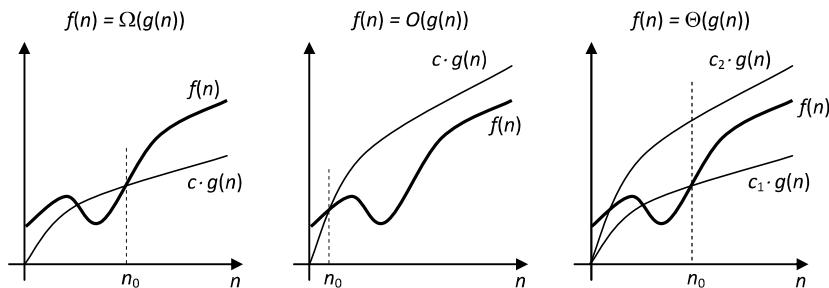
#### $\Theta$ -notacija

Neki algoritmi imaju istu složenost i u najboljem i u najgorem slučaju. Na primjer, ispis niza ima linearnu složenost i u najgorem i u najboljem slučaju, tj.  $\Omega(n) = O(n)$ . Slično, i množenje matrica ima kubnu složenost i u najgorem i u najboljem slučaju, tj.  $\Omega(n^3) = O(n^3)$ . Za algoritme koji imaju isti red složenosti i u najboljem i u najgorem slučaju koristi se  $\Theta$ -notacija (*teta*-notacija).

**Definicija:** Ako postoje pozitivne realne konstante  $c_1$  i  $c_2$  te prirodan broj  $n_0$ , takvi da za funkcije  $f$  i  $g$  važi  $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$  za svaki prirodan broj  $n > n_0$ , onda pišemo  $f(n) = \Theta(g(n))$  i čitamo "f je teta od g".

Za algoritme koji imaju isti red složenosti i u najboljem i u najgorem slučaju važi  $\Omega(g(n)) = O(g(n)) = \Theta(g(n))$ . Tako za algoritam množenja matrica možemo pisati  $\Omega(n^3) = O(n^3) = \Theta(n^3)$ .

Definicije  $\Omega$ -,  $O$ - i  $\Theta$ -notacije ilustrovane su na sl. 4.1.



Slika 4.1: Ilustracija definicije  $\Omega$ -,  $O$ - i  $\Theta$ -notacije

## 4.2. Osnovni pojmovi o sortiranju

Sortiranje je operacija nad kolekcijom podataka, kojom se elementi te kolekcije iz neuređenog rasporeda (**nesortirana kolekcija**) dovode u uređeni raspored (**sortirana kolekcija**). Ciljni poredak u kolekciji može biti:

- **rastući** (engl. *ascending*):
  - **stogo rastući**:  $niz[i] < niz[i + 1]$ ,
  - **monoton rastući (neopadajući)**:  $niz[i] \leq niz[i + 1]$ ;
- **opadajući** (engl. *descending*):
  - **stogo opadajući**:  $niz[i] > niz[i + 1]$ ,
  - **monoton opadajući (nerastući)**:  $niz[i] \geq niz[i + 1]$ .

Sortiranje je veoma česta operacija nad kolekcijama podataka, a sprovodi radi ostvarivanja sljedećih **ciljeva**:

- **efikasnije pretraživanje** (pronalaženje željenog podatka u kolekciji),
- jednostavnija **provjera jednakosti kolekcija**,
- **sistemizovan prikaz** sadržaja kolekcije (rangiranje elemenata) itd.

### 4.2.1. Klasifikacije sortiranja

Postupke sortiranja možemo klasifikovati prema različitim kriterijumima, npr. prema mjestu sortiranja, prema načinu manipulacije podacima, prema stabilnosti, prema tehnički sortiranja, prema efikasnosti itd.

#### Klasifikacija prema mjestu sortiranja

Prema mjestu sortiranja razlikujemo **unutrašnje i spoljašnje sortiranje**.

**Unutrašnje sortiranje** predstavlja sortiranje kolekcije podataka koja se nalazi u operativnoj memoriji, kao što je npr. sortiranje niza ili ulančane liste (vrsta dinamičke strukture podataka).

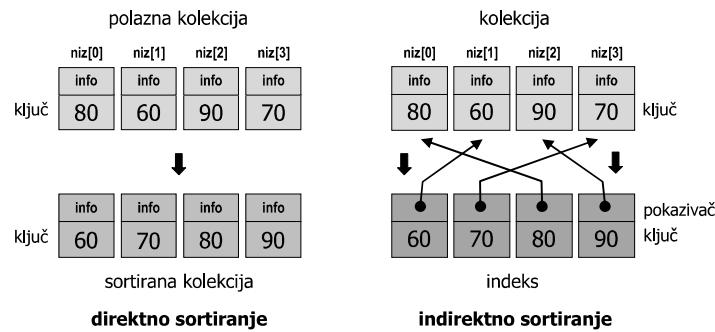
**Spoljašnje sortiranje** predstavlja sortiranje kolekcije podataka koja se nalazi u spoljašnjoj memoriji, tj. sortiranje podataka u datoteci.

#### Klasifikacija prema načinu manipulacije podacima

Prema načinu manipulacije podacima u kolekciji razlikujemo **direktno i indirektno sortiranje**.

**Direktno sortiranje** sprovodi se direktno na kolekciji i uključuje promjenu rasporeda elemenata u polaznoj kolekciji. Direktno sortiranje ilustrovano je lijevo na sl. 4.2. Kao primjer dat je niz slogova, pri čemu svaki slog ima informacioni sadržaj i ključ. Kolekcija se sortira na osnovu vrijednosti ključa. Ključ, u opštem slučaju, može biti reprezentovan jednim podatkom (npr. broj indeksa studenta) ili sa više podataka (npr. prezime studenta i ime studenta).

**Indirektno sortiranje** kolekcije ne vrši se promjenom rasporeda njenih elemenata, nego se formira pomoćna uređena (sortirana) kolekcija ključeva, koja se naziva **indeks** i koja sadrži pokazivače koji omogućavaju indirektan pristup podacima u nesortiranoj kolekciji. Indirektno sortiranje primjenjuje se u situacijama kada je cijena direktnog sortiranja previsoka – tipično kad veličina podataka u polaznoj kolekciji nije zanemarljiva i/ili kad je pristup elementima spor (sortiranje datoteka). Indirektno sortiranje ilustrovano je desno na sl. 4.2.

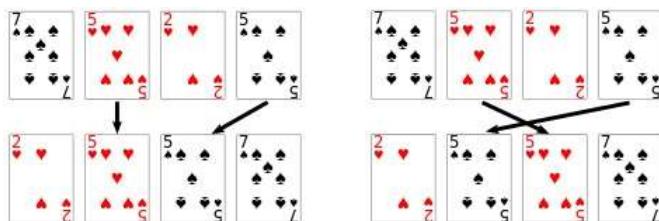


Slika 4.2: Ilustracija direktnog (lijevo) i indirektnog (desno) sortiranja

### Klasifikacija prema stabilnosti

Prema stabilnosti uzajamnog poretku elemenata sa istim vrijednostima ključa, razlikujemo **stabilno sortiranje** (engl. *stable sort*) i **nestabilno sortiranje** (engl. *unstable sort*). Kažemo da je sortiranje stabilno ako se prilikom sortiranja elemenata sa istom vrijednošću ključa ne mijenja njihov međusobni raspored u kolekciji. U protivnom, kažemo da je sortiranje nestabilno. Princip stabilnosti ilustrovan je na sl. 4.3 – lijevo je ilustrovano stabilno sortiranje (karta 5♥ se u polaznoj kolekciji nalazi ispred karte 5♦, a isti raspored je i u sortiranoj kolekciji nakon sortiranja po vrijednosti), a desno nestabilno sortiranje (nakon sortiranja, raspored karata 5♦ i 5♥ obrnut je u odnosu na polaznu kolekciju).

Treba uočiti da je stabilnost zagaranovana ako su svi ključevi u kolekciji različiti ili ako ključ čine svi podaci. Primjenjujući princip proširenja ključa dodatnim atributima (npr. dodavanje u ključ rednog broja podatka u kolekciji), nestabilni algoritmi mogu da se prevedu u stabilne. Proširenje ključa može da smanjuje efikasnost sortiranja (dodatno vrijeme i/ili memorija).



Slika 4.3: Stabilno (lijevo) i nestabilno (desno) sortiranje

### Klasifikacija prema tehnički sortiranja

Prema tehnički sortiranja razlikujemo **sortiranja zasnovana na poređenju elemenata kolekcije** (engl. *comparison sort*) i **sortiranja koja nisu zasnovana na poređenju elemenata kolekcije** (engl. *non-comparison sort*).

**Tehnike zasnovane na poređenju** elemenata (ključeva) su veoma često korištene tehnike. Ovim tehnikama ne može da se postigne manja vremenska složenost od  $O(n \log n)$  u najgorem slučaju. Najpoznatije metode su:

- **metode selekcije**

*Osnovni princip:* selektuje se odgovarajući element iz neuređenog dijela kolekcije i stavlja na kraj uređenog dijela kolekcije.

*Predstavnici:* **Select(ion)-sort.**

- **metode umetanja**

*Osnovni princip:* po jedan element iz neuređenog dijela kolekcije umeće se u uređeni dio kolekcije.

*Predstavnici:* **Insert(ion)-sort, Shell-sort.**

- **metode zamjene**

*Osnovni princip:* zamjena mjesta dva podatka koji nisu u odgovarajućem poretku.

*Predstavnici:* **Bubble-sort, Quick-sort.**

- **metode spajanja**

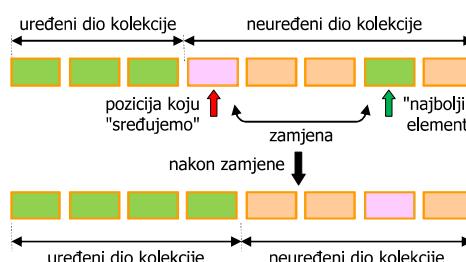
*Osnovni princip:* spajanje uređenih parcijalnih kolekcija.

*Predstavnici:* **Merge-sort.**

**Tehnikama koje nisu zasnovane na poređenju** elemenata (ključeva) može da se postigne i manja vremenska složenost od  $O(n \log n)$  u najgorem slučaju. Neke tehnike iz ove klase su **sortiranje brojanjem** (engl. *counting-sort*) i **radiks-sort** (engl. *radix-sort*).

## 4.3. Metode selekcije

Osnovni princip metoda selekcije jeste izbor odgovarajućeg elementa iz neuređenog dijela kolekcije i stavljanje na kraj uređenog dijela kolekcije, kao što je ilustrovano na sl. 4.4. U ovom odjeljku prikazana je **select(ion)-sort** tehnika, kao najpoznatija iz ove grupe.



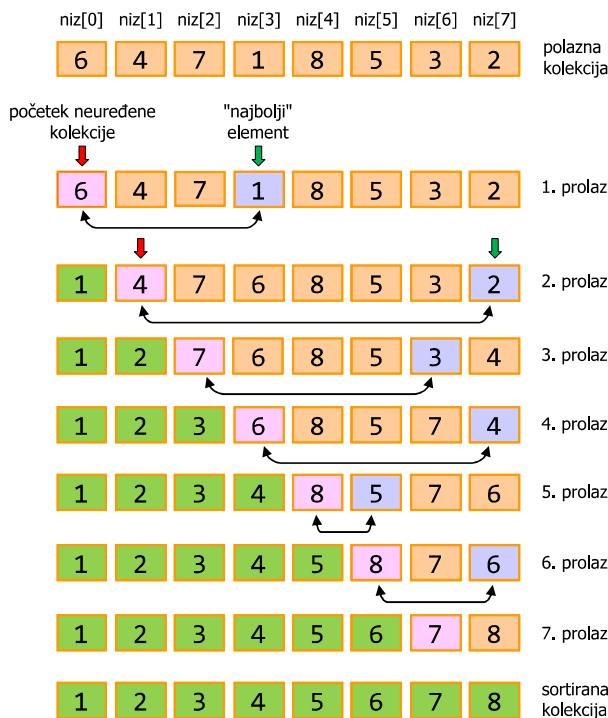
Slika 4.4: Ilustracija principa selekcije

### 4.3.1. Selection-sort

#### Osnovna verzija algoritma

1. pronađi odgovarajući element u kolekciji (najmanji ili najveći, zavisno od željenog rasporeda) i zamijeni ga sa prvim u kolekciji;
2. nastavi proceduru sa preostalim neuređenim dijelom kolekcije (pronađi odgovarajući element i zamijeni ga sa prvim elementom u neuređenom dijelu kolekcije) sve dok postoji neuređeni dio kolekcije.

Postupak sortiranja kolekcije u rastućem poretku ilustrovan je na sl. 4.5. Na početku je kolekcija neuređena. U svakom prolazu pronalazi se najmanji element u neuređenom dijelu kolekcije i zamijeni (ako postoji potreba) sa početnim elementom neuređenog dijela kolekcije. U prvom prolazu, minimalni element je  $niz[3]$  pa je izvršena zamjena  $niz[0] \leftrightarrow niz[3]$ , čime je "sredena" pozicija  $niz[0]$ . U drugom prolazu, minimalni element je  $niz[7]$  pa je izvršena zamjena  $niz[1] \leftrightarrow niz[7]$ . Nakon drugog prolaza, uređeni dio kolekcije čine  $niz[0]$  i  $niz[1]$ . Na isti način, postupak je nastavljen dalje. Treba uočiti da u sedmom prolazu nije došlo do zamjene, jer se vrijednost 7 već nalazi na odgovarajućoj poziciji (nema manje vrijednosti u neuređenom dijelu kolekcije). Nakon što se "sredi" pretposljednja pozicija, na posljednjoj poziciji u kolekciji nalazi se najveća vrijednost i postupak sortiranja je završen.



Slika 4.5: Ilustracija selection-sort metode

### Implementacija

---

```

void SelectionSort (<tip> niz[], int n)
{
    int i, j, min;
    for (i=0; i<n-1; i++)
    {
        for (min=i, j=i+1; j<n; j++)
            if (niz[j]<niz[min]) min=j;
        if (min!=i)
        {
            <tip> pom=niz[i];
            niz[i]=niz[min];
            niz[min]=pom;
        }
    }
}

```

---

Funkcija SelectionSort prima adresu niza podataka odgovarajućeg tipa (u konkretnoj implementaciji treba `<tip>` zamijeniti odgovarajućim stvarnim identifikatorom, npr. `int`) i broj podataka u nizu.

Spoljašnja petlja određuje granice uređenog dijela kolekcije. Kontrolna promjenljiva `i` pokazuje prvu poziciju iza kraja uređenog dijela kolekcije, odnosno poziciju koju "sređujemo". Na početku je `i=0` (sređujemo poziciju 0); kad sredimo poziciju 0, sređujemo poziciju 1; i tako dalje, sve do pretposljednje pozicije (sve dok je `i<n-1`).

U svakom ciklusu spolašnje petlje sređujemo poziciju koju pokazuje kontrolna promjenljiva `i`. Prvo se pronađe minimalni podatak u neuređenom dijelu kolekcije – iza pozicije `i` do kraja kolekcije (od pozicije `i+1` do pozicije `n-1`), što se realizuje unutrašnjom petljom. Na kraju prolaza, promjenljiva `min` sadrži poziciju minimuma. Ako je pronađen minimum (`min!=i`), vrši se zamjena vrijednosti (`niz[i]↔niz[min]`).

Prikazana implementacija omogućava sortiranje niza u neopadajućem redoslijedu. Za sortiranje u nerastućem redoslijedu, neophodno je operator `<` zamijeniti operatorom `>`.

Treba uočiti da se u prikazanoj implementaciji porede podaci elementarnih tipova, jer se operatori poređenja (`<`, `<=`, `>`, `>=`) mogu primjenjivati samo nad elementarnim podacima. Ako su elementi niza slogovi (strukture), tada se vrši poređenje odgovarajućih elemenata strukture koji predstavljaju ključ, npr. `niz[j].kljuc<niz[min].kljuc`.

Treba uočiti da se korištenjem operatora `<` ostvaruje stabilno sortiranje – u slučaju elemenata sa istim vrijednostima ključa, biće sačuvan njihov uzajamni redoslijed. Ako bi se koristio operatator `<=`, bio bi narušen princip stabilnosti, jer bi došlo do promjene uzajamnog rasporeda elemenata sa istim vrijednostima ključa.

### Analiza složenosti

Unutrašnja petlja sadrži: dvije dodjele početnih vrijednosti ( $\min$  i  $j$ ),  $n-i-1$  provjeru uslova za ponavljanje ciklusa,  $n-i-2$  inkrementovanja brojača te  $n - i - 2$  izvršavanja ciklusa. U svakom ciklusu, u najgorem slučaju, biće provjeren uslov i dodijeljena nova vrijednost promjenljivoj  $\min$ , što je ukupno  $2(n - i - 2)$  operacija. Dakle, ukupan broj koraka koji sadrži unutrašnja petlja iznosi:

$$T'(n) = 2 + (n - i - 1) + (n - i - 2) + 2(n - i - 2) = 4n - 4i - 5.$$

Svaki ciklus spoljašnje petlje sadrži jednu unutrašnju petlju i, u najgorem slučaju, jednu provjeru uslova i tri naredbe dodjele, što znači da, u najgorem slučaju, ukupan broj koraka u svakom ciklusu spoljašnje petlje iznosi:

$$T''(n) = T'(n) + 4 = 4n - 4i - 1.$$

U  $i$ -tom koraku spoljašnje petlje ciklus sadrži  $T''(n)$  koraka, pa je ukupan broj koraka u svim ciklusima spoljašnje petlje:

$$\begin{aligned} T'''(n) &= \sum_{i=0}^{n-2} T''(n) = \sum_{i=0}^{n-2} (4n - 4i - 1) = 4 \sum_{i=0}^{n-2} n - 4 \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 \\ T'''(n) &= 4n(n - 1) - 4 \frac{(n - 2)(n - 1)}{2} - (n - 1) = 2n^2 + n - 5. \end{aligned}$$

Spoljašnja petlja sadrži: jednu dodjelu početne vrijednosti,  $n$  provjera uslova,  $n - 1$  inkrementovanja brojača te sve korake ciklusa. Ukupan broj koraka spoljašnje petlje je:

$$T(n) = 1 + n + (n - 1) + T'''(n) = 2n + (2n^2 + n - 5) = 2n^2 + 3n - 5.$$

Red funkcije složenosti je dva i važi

$$T(n) = O(n^2).$$

Dakle, *selection-sort* ima kvadratnu složenost u najgorem slučaju.

Kao što je ranije navedeno, za procjenu složenosti nije neophodno da se funkcija složenosti tačno odredi, nego samo da se odredi njen red. Budući da implementacija sadrži ugnježđenu petlju, odmah smo mogli zaključiti da je red funkcije složenosti jednak dva.

Treba uočiti da na složenost algoritma najviše utiče određivanje minimuma u neuređenom dijelu kolekcije. U svakom prolazu kroz neuređeni dio, uvijek se izvrše sva poređenja. Zbog toga, složenost ne zavisi od inicijalnog rasporeda – čak i da je kolekcija inicijalno sortirana, opet bismo imali sva poređenja (samo ne bismo imali promjene promjenljive  $\min$ ).

Određeno ubrzanje moguće je postići istovremenim sortiranjem sa oba kraja. Tada bismo u svakom prolazu tražili i minimum i maksimum, pri čemu bismo minimum stavljali na početak, a maksimum na kraj neuređenog dijela kolekcije. Tako bi se uređeni dijelovi kolekcije širili s krajeva prema sredini. I tada bi složenost bila kvadratna, jer bi broj poređenja i dalje bio isti.

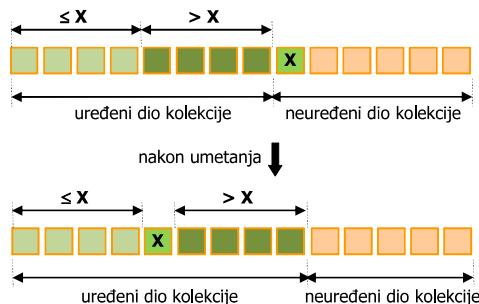
## 4.4. Metode umetanja

Osnovni princip metoda umetanja jeste uzimanje po jednog elementa iz neuređenog dijela kolekcije i umetanje na odgovarajuće mjesto u uređenom dijelu kolekcije. U ovom odjeljku prikazane su najpoznatije tehnike iz ove grupe: **insert(ion)-sort** i **Shell-sort**.

### 4.4.1. Insert(ion)-sort

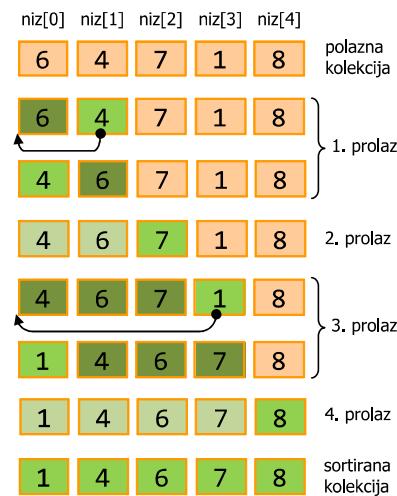
#### Osnovna verzija algoritma

Osnovna ideja *insertion-sort* algoritma dobro je poznata i često se primjenjuje za ređanje karata u društvenim igrama (sl. 4.6). U svakom koraku, prvi element ( $X$ ) iz nesortiranog dijela kolekcije ubacuje se na odgovarajuće mjesto u sortiranom dijelu kolekcije. Kao što se vidi na sl. 4.6, uređeni dio kolekcije sadrži elemente koji su manji ili jednaki elementu  $X$  i elemente koji su veći od  $X$ . Ako se element  $X$  ubaci između ova dva uređena dijela, dobijemo uređenu kolekciju koja sadrži  $X$ , čime se uređeni dio kolekcije povećava, a neuređeni dio smanjuje za  $X$ . Ubacivanje  $X$  realizuje se pomjeranjem elemenata koji su veći od  $X$  za jedno mjesto prema kraju, nakon čega se na upražnjeno mjesto smješta  $X$ .



Slika 4.6: Ilustracija principa umetanja

Postupak sortiranja kolekcije u rastućem poretku ilustrovan je na sl. 4.7. Na početku sortiranja, uređeni dio kolekcije sadrži samo prvi element niza (6), a svi ostali elementi pripadaju neuređenom dijelu kolekcije. U prvom prolazu, prvi element iz neuređenog dijela kolekcije (4) ubačen je ispred elementa 6 (jer je  $4 < 6$ ). Na kraju prvog prolaza uređeni dio kolekcije je  $\{4 \ 6\}$ . U drugom prolazu nije bilo premještanja prvog elementa iz neuređenog dijela kolekcije (7), jer nije manji ni od jednog elementa u uređenom dijelu kolekcije. U trećem prolazu, prvi element iz neuređenog dijela kolekcije (1) manji je od svih elemenata u uređenom dijelu kolekcije pa su svi elementi uređenog dijela kolekcije pomjereni jedno mjesto prema kraju, a na upražnjeno mjesto ubaćena vrijednost 1. U četvrtom prolazu nije bilo premještanja (slično kao u drugom prolazu).

Slika 4.7: Ilustracija *insertion-sort* metode

### Implementacija

---

```
void InsertionSort (<tip> niz[], int n)
{
    int i, j;
    for (i=1; i<n; i++)
    {
        <tip> x=niz[i];
        for (j=i; j>0 && x<niz[j-1]; j--)
            niz[j]=niz[j-1];
        niz[j]=x;
    }
}
```

---

Funkcija `InsertionSort` prima adresu niza podataka odgovarajućeg tipa (u konkretnoj implementaciji treba `<tip>` zamijeniti odgovarajućim stvarnim identifikatorom, npr. `int`) i broj podataka u nizu.

Spoljašnja petlja određuje granice neuređenog dijela kolekcije – kontrolna promjenljiva `i` broji prolaze kroz kolekciju i u svakom koraku pokazuje na početni element u neuređenom dijelu kolekcije, kojeg ubacujemo u uređeni dio kolekcije. Početna vrijednost kontrolne promjenljive je `i=1`, jer neuređeni dio kolekcije počinje od pozicije 1 (`niz[0]` pripada uređenom dijelu kolekcije). Postupak se ponavlja sve dok ima elemenata u neuređenom dijelu kolekcije (sve dok je `i<n`).

U svakom ciklusu spoljašnje petlje ubacujemo početni element ( $niz[i]$ ) iz neuređenog dijela kolekcije u uređeni dio kolekcije. Pomoćna promjenljiva služi da privremeno sačuvamo vrijednost elementa  $niz[i]$ , kako bismo mogli pomjeriti elemente iz uređenog dijela kolekcije. Unutrašnja petlja služi za

pomjeranje uređenog dijela kolekcije, element po element, od posljednjeg elementa pa sve dok ima elemenata u uređenom dijelu i sve dok su oni veći od vrijednosti koju treba da ubacimo. Na kraju se na upražnjeno mjesto smješta vrijednost  $x$ .

Prikazana implementacija omogućava sortiranje niza u neopadajućem redoslijedu. Za sortiranje u nerastućem redoslijedu, neophodno je operator  $<$  zamijeniti operatom  $>$ .

Treba uočiti da se u prikazanoj implementaciji porede podaci elementarnih tipova, jer se operatori poređenja ( $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ) mogu primjenjivati samo nad elementarnim podacima. Ako su elementi niza slogovi (strukture), tada se vrši poređenje odgovarajućih elemenata strukture koji predstavljaju ključ, npr.  $x.\text{kljuc} < \text{niz}[j-1].\text{kljuc}$ .

Treba uočiti da se korištenjem operatora  $<$  ostvaruje stabilno sortiranje – u slučaju elemenata sa istim vrijednostima ključa, biće sačuvan njihov uzajamni redoslijed. Ako bi se koristio operator  $\leq$ , bio bi narušen princip stabilnosti, jer bi došlo do promjene uzajamnog rasporeda elemenata sa istim vrijednostima ključa.

### Analiza složenosti

Unutrašnja petlja sadrži: jednu dodjelu početne vrijednosti, maksimalno  $i + 1$  provjera uslova za ponavljanje ciklusa, maksimalno  $i$  inkrementovanja brojača te maksimalno  $i$  izvršavanja ciklusa. Dakle, ukupan broj koraka koji sadrži unutrašnja petlja iznosi:

$$T'(n) = 1 + (i + 1) + i + i = 3i + 2.$$

Svaki ciklus spoljašnje petlje sadrži dvije naredbe dodjele i unutrašnju petlju pa, u najgorem slučaju, ukupan broj koraka u svakom ciklusu spoljašnje petlje iznosi:

$$T''(n) = T'(n) + 2 = 3i + 4.$$

U  $i$ -tom koraku spoljašnje petlje ciklus sadrži  $T''(n)$  koraka, pa je ukupan broj koraka u svim ciklusima spoljašnje petlje:

$$\begin{aligned} T'''(n) &= \sum_{i=1}^{n-1} T''(n) = \sum_{i=1}^{n-1} (3i + 4) = 3 \sum_{i=1}^{n-1} i + 4 \sum_{i=1}^{n-1} 1 \\ T'''(n) &= 3 \frac{n(n-1)}{2} + 4(n-1) = \frac{3n^2 + 5n - 8}{2}. \end{aligned}$$

Spoljašnja petlja sadrži jednu dodjelu početne vrijednosti,  $n$  provjera uslova,  $n - 1$  inkrementovanja brojača te sve korake ciklusa. Ukupan broj koraka spoljašnje petlje je:

$$T(n) = 1 + n + (n - 1) + T'''(n) = 2n + \frac{3n^2 + 5n - 8}{2} = \frac{3n^2 + 9n - 8}{2}.$$

Red funkcije složenosti je dva i važi

$$T(n) = O(n^2).$$

Dakle, *insertion-sort* ima kvadratnu složenost u najgorem slučaju. Kao što je ranije navedeno, za procjenu složenosti nije neophodno da se funkcija složenosti tačno odredi, nego samo da se odredi njen red. Budući da implementacija sadrži ugnježđenu petlju, odmah smo mogli zaključiti da je red funkcije složenosti jednak dva.

Iz primjera ilustrovanog na sl. 4.7, uočljivo je da postoje prolazi u kojima nema ubacivanja. To se dešava ako se prvi element u neuređenom dijelu kolekcije nalazi na odgovarajućem mjestu. Tada se unutrašnja petlja svodi na jednu provjeru uslova, a ciklus spoljašnje petlje ima samo tri koraka. Ako bi kompletan polazna kolekcija već bila sortirana, nijedan element ne bi bio pomjeren pa bi ukupan broj koraka spoljašnje petlje bio:

$$T(n) = 1 + n + (n - 1) + T'''(n) = 2n + 3,$$

što znači da *insertion-sort* ima linearnu složenost u najboljem slučaju, tj.

$$T(n) = \Omega(n).$$

Prethodna analiza pokazuje da brzina sortiranja zavisi od inicijalnog rasporeda – što je inicijalni raspored bliži željenom poretku, to je sortiranje brže. Da zaključimo, *insertion-sort* je veoma povoljan za male i skoro uređene nizove.

#### 4.4.2. Shell-sort

Osnovni nedostatak *insertion-sort* algoritma jeste veliki broj pomjerenja elemenata uređenog dijela kolekcije radi ubacivanja elementa iz neuređenog dijela kolekcije na početak uređenog dijela (u najgorem slučaju). *Shell-sort*<sup>1</sup> predstavlja poboljšani *insertion-sort* metod, koji smanjuje broj pomjerenja. Ovaj algoritam pripada grupi algoritama za brzo sortiranje.

##### Osnovna ideja algoritma:

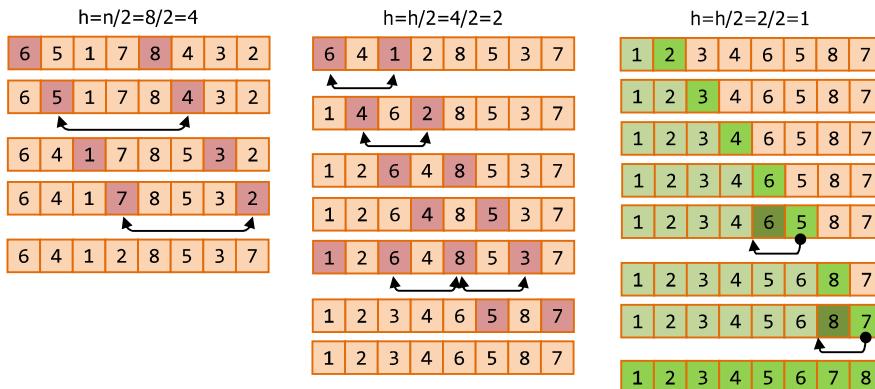
- u prvom prolazu kroz kolekciju porede se (i po potrebi zamjenjuju) ekvidistantri elementi – npr. elementi razmagnuti za  $h = n/2$  (npr. za  $n = 8$  imamo  $h = 4$  pa poredimo  $5 \leftrightarrow 1, 6 \leftrightarrow 2$  itd.);
- u svakom narednom prolazu kroz kolekciju distanca se smanjuje (npr.  $h = h/2$ ) i postupak nastavlja sve dok je  $h \geq 1$ .

Postupak sortiranja kolekcije u rastućem poretku ilustrovan je na sl. 4.8. Broj podataka u nizu je  $n = 8$  pa je distanca s kojom se porede elementi  $h = n/8 = 4$ . U prvom prolazu porede se elementi  $niz[0]$  i  $niz[4]$ , čije su vrijednosti 6 i 8. Pošto su 6 i 8 u odgovarajućem poretku, nema pomjerenja. U drugom prolazu porede se elementi  $niz[1]$  i  $niz[5]$ . Pošto je  $4 < 5$ , vrši se pomjerenje (zamjena vrijednosti  $4 \leftrightarrow 5$ ). U trećem prolazu nema pomjerenja, a u četvrtom dolazi do pomjerenja (zamjena vrijednosti  $2 \leftrightarrow 7$ ). Konačno, nakon svih prolaza kroz niz sa distancom  $h = 4$ , imamo kolekciju  $\{6, 4, 1, 2, 8, 5, 3, 7\}$ .

<sup>1</sup> *Shell-sort* nosi naziv po svom autoru, američkom naučniku Donaldu Shelliu (1924-2015).

Nakon toga, distanca se smanjuje ( $h = h/2 = 2$ ) i postupak ponavlja. Prvo se porede elementi  $niz[0]$  i  $niz[2]$ , gdje dolazi do pomjeranja i zamjene ( $6 \leftrightarrow 1$ ). Zatim se porede elementi  $niz[1]$  i  $niz[3]$ , gdje dolazi do pomjeranja i zamjene ( $4 \leftrightarrow 2$ ). Slijede dva prolaza bez pomjeranja. Nakon toga ide poređenje  $niz[6]$  i  $niz[4]$  i pomjeranje vrijednosti 8, pa poređenje  $niz[4]$  i  $niz[2]$  i pomjeranje vrijednosti 6. Tu se pomjeranje zaustavlja (jer je  $3 > 1$ ), a vrijednost 3 se ubacuje na poziciju  $niz[2]$ . Nakon toga slijedi prolaz bez pomjeranja (jer je  $7 > 5$ ). Nakon svih prolaza sa distancom  $h = 2$ , imamo kolekciju  $\{1, 2, 3, 4, 6, 5, 8, 7\}$ .

Konačno se distanca smanjuje na jediničnu ( $h = h/2 = 1$ ), čime se postupak svodi na originalni *insertion-sort* postupak. Kao što se vidi sa sl. 4.8, inicijalni prolazi kroz niz sa većim distancama ( $h > 1$ ) u značajnoj mjeri su preuredili polaznu sekvencu i smanjili potrebu za premještanjem susjednih elemenata. Tako su, u konkretnom slučaju, bila neophodna samo dva pomjeranja ( $6 \leftrightarrow 5$  i  $8 \leftrightarrow 7$ ).



Slika 4.8: Ilustracija *shell-sort* metode

### Implementacija

---

```

void ShellSort (<tip> niz[], int n)
{
    int i, j, h;
    for (h=n/2; h>0; h/=2)
    {
        for (i=h; i<n; i++)
        {
            <tip> x=niz[i];
            for (j=i; j>=h && x<niz[j-h]; j-=h)
                niz[j]=niz[j-h];
            niz[j]=x;
        }
    }
}

```

---

Funkcija `ShellSort` prima adresu niza podataka odgovarajućeg tipa (u konkretnoj implementaciji treba `<tip>` zamijeniti stvarnim identifikatorom tipa) i broj podataka u nizu.

Spoljašnjom petljom omogućavaju se višestruki prolazi kroz sekvencu sa različitim distancama, pri čemu kontrolna promjenljiva  $h$  predstavlja distancu. Inicijalna distanca je  $h=n/2$ , a postupak se ponavlja sve do jedinične distance ( $h>0$ ). Nakon svakog ciklusa, distanca se polovi ( $h/=2$ ).

U svakom ciklusu spoljašnje petlje (za konkretno  $h$ ), vrše se poređenja, i po potrebi pomjeranja, ekvidistantnih elemenata. Ovo je veoma slično osnovnom *insertion-sort* algoritmu, samo što se poređenje i pomjeranje vrši sa distancom. Zbog toga kontrolna promjenljiva  $i$ , koja omogućava prolaze kroz kolekciju, prolazi sve vrijednosti od  $h$  do  $n-1$  (a ne od 1 do  $n-1$ , kao kod osnovnog algoritma). Poređenje i pomjeranje elemenata sa distancom  $h$  realizuje se petljom koja je kontrolisana promjenljivom  $j$ . Treba uočiti da je za  $h=1$ , ciklus spoljašnje petlje ekvivalentan osnovnom *insertion-sort* algoritmu.

Prikazana implementacija omogućava sortiranje niza u neopadajućem redoslijedu. Za sortiranje u nerastućem redoslijedu, neophodno je operator `<` zamijeniti operatorom `>`. Pored toga, u prikazanoj implementaciji porede se podaci elementarnih tipova. Ako bi elementi niza bili slogovi (strukture), tada bi trebalo porediti odgovarajuće elemente strukture koji predstavljaju ključ, npr. `x.kljuc<niz[j-h].kljuc`.

Treba uočiti da se korištenjem operatora `<` ostvaruje stabilno sortiranje – u slučaju elemenata sa istim vrijednostima ključa, biće sačuvan njihov uzajamni redoslijed. Ako bi se koristio operator  `$\leq$` , bio bi narušen princip stabilnosti, jer bi došlo do promjene uzajamnog rasporeda elemenata sa istim vrijednostima ključa.

### Složenost algoritma

*Shell-sort* pripada grupi algoritama za brzo sortiranje. Efikasnost algoritma zavisi od izabrane sekvence za  $h$ .

Za prikazanu implementaciju, koja se zasniva na polovljenju distance ( $h = h/2$ ), u najgorem slučaju složenost je:

$$T(n) = O(n^2).$$

Pogodnim izborom sekvence može da se smanji složenost algoritma i poveća efikasnost. *Hibbard-ovom* sekvencom ( $h = 2^k - 1, \dots, 7, 3, 1$ ), u najgorem slučaju postiže se složenost:

$$T(n) = O(n^{3/2}),$$

dok se *Sedgwick-ovom* sekvencom ( $h = \dots, 109, 41, 19, 5, 1$ ) u najgorem slučaju postiže:

$$T(n) = O(n^{4/3}).$$

## 4.5. Metode zamjene

Osnovni princip metoda umetanja jeste zamjena dva elementa kolekcije koji nisu u odgovarajućem poretku. U ovom odjeljku prikazane su najpoznatije tehnike iz ove grupe: **bubble-sort** i **quick-sort**.

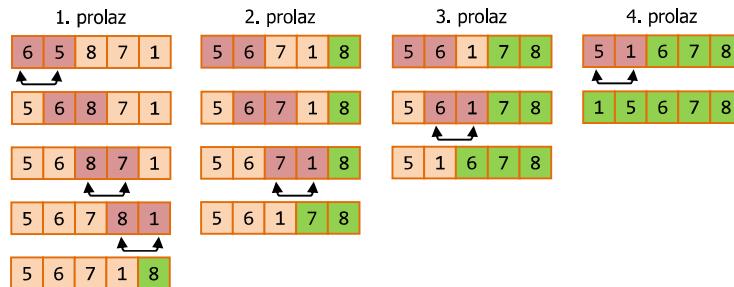
### 4.5.1. Bubble-sort

**Osnovna ideja bubble-sort algoritma** jeste **zamjena svaka dva susjedna elementa koji nisu u odgovarajućem poretku**. Ovaj metod još se naziva i **metod direktnе zamjene**.

U skladu sa osnovnom idejom, kreće se od početka kolekcije prema kraju i porede svaka dva susjedna elementa. Ako elementi nisu u odgovarajućem poretku, izvrši se zamjena. Tako, u jednom prolazu kroz kolekciju, najlakši (ili najteži, zavisno od željenog poretku) "isplova na površinu", tj. na početak uređenog dijela kolekcije.

U svakom prolazu kroz kolekciju, po jedan element prelazi iz neuređenog dijela u uređeni dio kolekcije, pri čemu neuređeni dio kolekcije sadrži početne elemente, a uređeni dio sadrži krajnje elemente kolekcije. Prolasci se ponavljaju sve dok u neuređenom dijelu ne ostane samo jedan element (on je sigurno na odgovarajućoj poziciji, jer je u prethodnom koraku došao na tu poziciju). Sortiranje može biti zaustavljeno i ako u nekom prolazu kroz kolekciju nije bilo zamjena, jer to znači da je kolekcija sortirana i da je sortiranje završeno.

Postupak sortiranja kolekcije u rastućem poretku ilustrovan je na sl. 4.9. U svakom prolazu, poređenje kreće od početnog elementa i završava krajnjim elementom u neuređenom dijelu kolekcije. U prvom prolazu bila su četiri poređenja uz tri zamjene. Nakon prvog prolaza, na kraju kolekcije pojavila se najveća vrijednost (8). U drugom prolazu bila su tri poređenja uz jednu zamjenu, nakon čega se na početku uređenog dijela kolekcije pojavila vrijednost 7 pa je uređeni dio kolekcije {7, 8}. U trećem prolazu bila su dva poređenja uz jednu zamjenu, nakon čega je uređeni dio kolekcije {6, 7, 8}. U četvrtom prolazu bilo je jedno poređenje koje je završilo zamjenom. Budući da je u neuređenom dijelu kolekcije ostao samo jedan podatak (1), sortiranje je završeno i rezultat sortiranja je uređena kolekcija {1, 5, 6, 7, 8}.



Slika 4.9: Ilustracija *bubble-sort* metoda

### Implementacija

---

```
void BubbleSort (<tip> niz[], int n)
{
    int i, j;
    for (i=n-1; i>0; i--)
    {
        for (j=0; j<i; j++)
            if (niz[j]>niz[j+1])
            {
                <tip> pom=niz[j];
                niz[j]=niz[j+1];
                niz[j+1]=pom;
            }
    }
}
```

---

Funkcija `BubbleSort` prima adresu niza podataka odgovarajućeg tipa (u konkretnoj implementaciji treba `<tip>` zamijeniti stvarnim identifikatorom tipa) i broj podataka u nizu.

Kontrolna promjenljiva `i` spoljašnje petlje u svakom koraku pokazuje krajnji element u neuređenom dijelu kolekcije (početna vrijednost je `n-1`, a krajnja 1). Svakim korakom neuređeni dio kolekcije smanjuje se za jedan element (`i--`).

Svakim ciklusom spoljašnje petlje realizuje se po jedan prolaz kroz niz. U svakom prolazu, poređenja i eventualne zamjene u neuređenom dijelu kolekcije realizuju se unutrašnjom petljom (kraj neuređenog dijela određen je kontrolnom promjenljivom `i`).

Prikazana implementacija omogućava sortiranje niza u neopadajućem redoslijedu. Za sortiranje u nerastućem redoslijedu, neophodno je operator `<` zamijeniti operatorom `>`. Pored toga, u prikazanoj implementaciji porede se podaci elementarnih tipova. Ako bi elementi niza bili slogovi (strukture), tada bi trebalo porebiti odgovarajuće elemente strukture koji predstavljaju ključ, npr. `niz[j].kljuc < niz[j+1].kljuc`. Korištenjem operatora `>` ostvaruje se stabilno sortiranje – u slučaju elemenata sa istim vrijednostima ključa, biće sačuvan njihov uzajamni redoslijed. Ako bi se koristio operator `>=`, bio bi naorušen princip stabilnosti, jer bi došlo do promjene uzajamnog rasporeda elemenata sa istim vrijednostima ključa.

### Analiza složenosti

Unutrašnja petlja sadrži: jednu dodjelu početne vrijednosti,  $i + 1$  provjera uslova za ponavljanje ciklusa,  $i$  inkrementovanja brojača te  $i$  ciklusa. Svaki ciklus unutrašnje petlje u najgorem slučaju sadrži četiri koraka (poređenje i tri dodjele), pa je ukupan broj koraka u spoljašnjem ciklusu:

$$T'(n) = 1 + (i + 1) + i + 4i = 6i + 2.$$

Spoljašnja petlja sadrži jednu dodjelu početne vrijednosti,  $n$  provjera uslova,  $n - 1$  inkrementovanja brojača te sve korake ciklusa. Ukupan broj koraka spoljašnje petlje je:

$$\begin{aligned} T(n) &= 1 + n + (n - 1) + \sum_{i=1}^{n-1} (6i + 2) = 2n + 6 \sum_{i=1}^{n-1} i + 2 \sum_{i=1}^{n-1} 1 \\ T(n) &= 2n + 6 \frac{n(n-1)}{2} + 2(n-1) = 3n^2 + n - 2. \end{aligned}$$

Dakle, **bubble-sort ima kvadratnu složenost u najgorem slučaju** (obrnuto sortiran niz), tj.

$$T(n) = O(n^2).$$

U najboljem slučaju, inicijalna kolekcija je već sortirana. Tada bi već nakon prvog prolaza moglo da se konstatiše da nije bilo zamjena (neophodno je uvesti odgovarajuću identifikaciju da li je bilo zamjena). U tom slučaju, imali bismo samo jedan prolaz kroz niz, odnosno **linearnu složenost**, tj.

$$T(n) = \Omega(n).$$

Treba uočiti (sl. 4.9) da veći podaci s početka niza brzo idu prema kraju ("zečevi"), a da manji podaci s kraja niza sporo idu prema početku ("kornjače"). Brzo napredovanje elemenata s početka niza prema kraju podsjeća na kretanje balončića (engl. *bubble*) od dna prema vrhu pa otuda potiče i naziv *bubble-sort*.

#### 4.5.2. Quick-sort

*Quick-sort* je rekurzivni metod za sortiranje, koji slijedi strategiju "*podijeli pa osvoji*" (engl. *divide-and-conquer*). Ovaj algoritam pripada grupi algoritama za brzo sortiranje.

##### Osnovna ideja algoritma

- **osnovni slučaj:**  
broj elemenata u kolekciji 0 ili 1  $\implies$  prekini izvršavanje;
- **rekurzivni korak:**
  1. izaberi razdvojni element (**pivot**) u kolekciji;
  2. rasporedi ostale elemente iz kolekcije u dva podskupa:  
 $S_m = \{x \in \text{niz} \setminus \{\text{pivot}\} \mid x \leq \text{pivot}\},$   
 $S_v = \{x \in \text{niz} \setminus \{\text{pivot}\} \mid x \geq \text{pivot}\};$
  3. vrati sortirani niz:  $\{\text{QuickSort}(S_m), \text{pivot}, \text{QuickSort}(S_v)\}.$

**Pivot** razdvaja elemente kolekcije u dvije kolekcije – kolekciju manjih od pivota ( $S_m$ ) i kolekciju većih od pivota ( $S_v$ ). Izbor pivota nije jedinstveno određen, odnosno pivot može da se bira na različite načine.

### Implementacija

---

```
void QuickSort(<tip> niz[], int begin, int end)
{
    if (begin < end)
    {
        int pivot = split(niz, begin, end);
        QuickSort(niz, begin, pivot-1);
        QuickSort(niz, pivot+1, end);
    }
}
```

---

Funkcija QuickSort prima adresu niza podataka odgovarajućeg tipa te poziciju početka i poziciju kraja niza. Inicijalno se funkcija poziva sa

`QuickSort(niz, 0, n-1);`

gdje je  $n$  ukupan broj podataka u nizu. Pivot se određuje funkcijom `split`. Nakon što se odredi pivot, vrši se sortiranje podniza  $S_m$  (dio niza od pozicije `begin` do pozicije `pivot-1`) te sortiranje podniza  $S_v$  (dio niza od pozicije `pivot+1` do pozicije `end`).

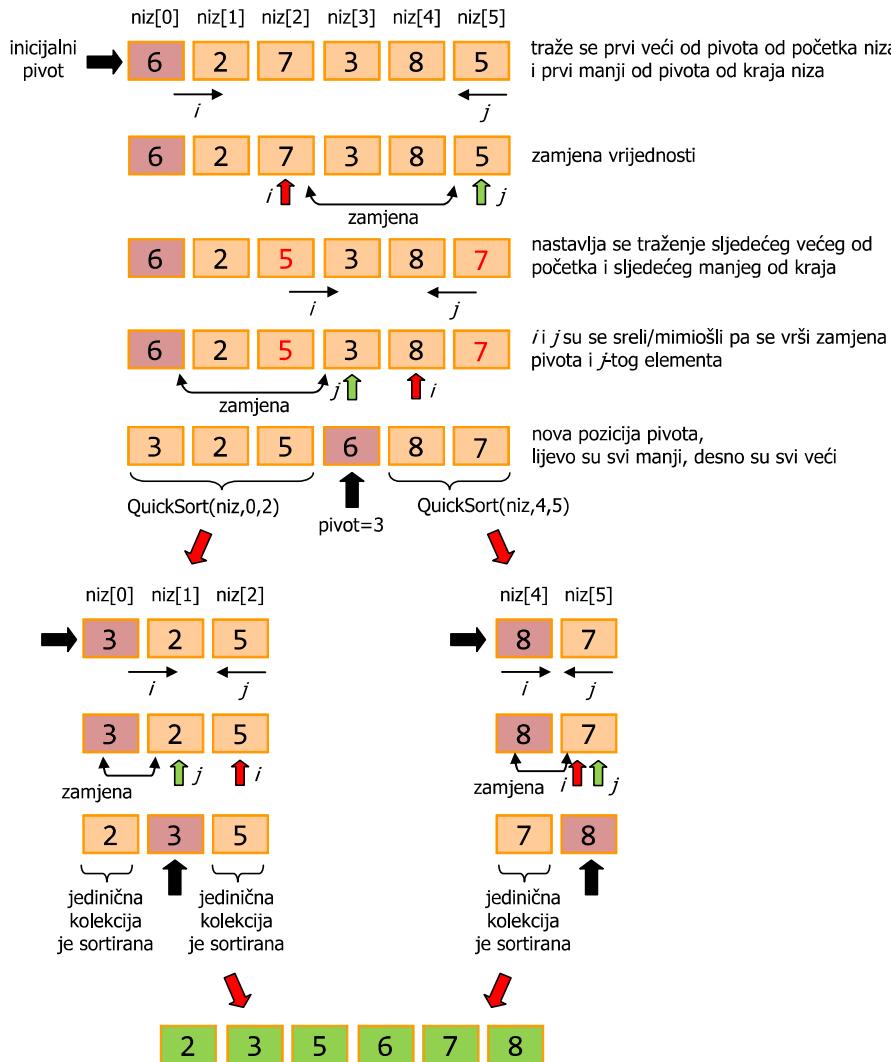
Kao što je ranije navedeno, funkcija `split`, odnosno izbor pivota može da se realizuje na različite načine. Implementacija u ovom slučaju traži sve veće od početnog elementa u nizu (uzet za pivota) sa lijeve strane i mijenja ih sa svim manjim od pivota sa desne strane – kad se sretnu granice, tu se postavi pivot.

---

```
int split(int niz[], int begin, int end)
{
    int i=begin, j=end;
    int pivot = niz[begin];
    while (i<j)
    {
        while (niz[i]<=pivot && i<j) i++;
        while (niz[j]>pivot) j--;
        if (i<j)
        {
            int pom=niz[i];
            niz[i]=niz[j];
            niz[j]=pom;
        }
    }
    niz[begin] = niz[j];
    niz[j] = pivot;
    return j;
}
```

---

Postupak sortiranja kolekcije u rastućem poretku ilustrovan je na sl. 4.10.



Slika 4.10: Ilustracija *quick-sort* metoda

### Složenost algoritma

Iako je složenost algoritma u najgorem slučaju kvadratna, *quick-sort* ipak pripada grupi algoritama za brzo sortiranje, jer je **prosječna složenost linearno-logaritamska**, tj.

$$T(n) = O(n \log n).$$

### Implementacija u standardnoj C biblioteci

Standardna biblioteka (zaglavljek `<stdlib.h>`) raspolaže funkcijom `qsort` koja implementira *quick-sort* algoritam. Prototip funkcije `qsort` je:

---

```
void qsort(void *niz, size_t n, size_t vel,
           int (*cmp)(const void*, const void*));
```

---

Argumenti funkcije `qsort` su:

- **niz** – adresa niza;
- **n** – broj podataka u nizu;
- **vel** – veličina svakog pojedinačnog podatka u nizu;
- **cmp** – adresa komparatorske funkcije (za poređenje elemenata niza).

---

#### Primjer 4.4:

Sortiranje niza korištenjem standardne funkcije `qsort`.

```
#include <stdio.h>
#include <stdlib.h>
int cmpInt (const void *a, const void *b)
{
    return *(int*)a - *(int*)b;
}
void pisiNiz(int niz[], int n)
{
    for (int i=0; i<n; i++)
        printf(" %d", niz[i]);
}
int main ()
{
    int niz[]={6, 2, 7, 3, 8, 5};
    printf("Prije sortiranja:"); pisiNiz(niz, 6);
    qsort(niz, 6, sizeof(int), cmpInt);
    printf("\nNakon sortiranja:"); pisiNiz(niz, 6);
    return 0;
}
```

Rezultat izvršavanja programa prikazan je na slici.

```
Prije sortiranja: 6 2 7 3 8 5
Nakon sortiranja: 2 3 5 6 7 8
```

Treba uočiti način na koji se definije komparatorska funkcija. U konkretnom slučaju, definisana je komparatorska funkcija za poređenje cijelih brojeva, koja prima dvije adrese, zatim te adrese konvertuje u adrese podataka stvarnog tipa (`int`) i indirektno pristupa podacima. Prilikom poređenja, komparatorska funkcija treba da vrati rezultat `-1` ako je prvi podatak manji od drugog, nulu ako su podaci jednaki, a `1` ako je prvi podatak veći od drugog. Na taj način realizuje se sortiranje u neopadajućem poretku (za sortiranje u nerastućem poretku, trebalo bi zamijeniti operande). Na isti način može da se definije komparator za druge tipove.

## 4.6. Metode spajanja

Osnovni princip metoda spajanja jeste spajanje uređenih kolekcija u jednu uređenu kolekciju. U ovom odjeljku prikazana je najpoznatija tehnika iz ove grupe: **merge-sort**.

### 4.6.1. Merge-sort

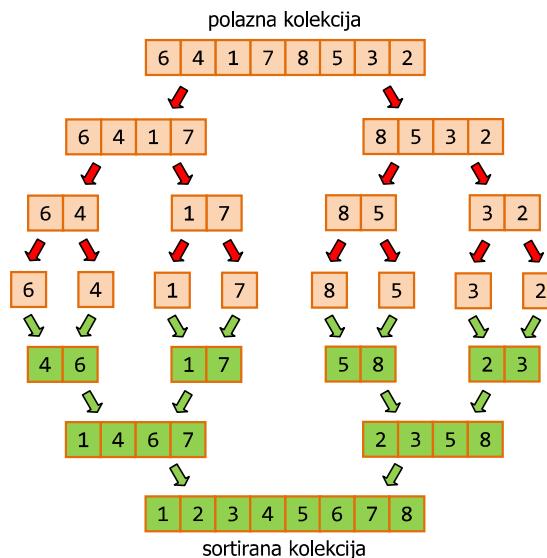
*Merge-sort* je rekurzivni metod za sortiranje. I ovaj algoritam pripada grupi algoritama za brzo sortiranje.

**Osnovna ideja algoritma** jeste da se nesortirana kolekcija podijeli na dva (pod)jednaka dijela, koji se dalje dijele, dok se ne dođe do jedinične kolekcije (jedinična kolekcija je sortirana). Na kraju se dvije sortirane kolekcije spajaju u jednu sortiranu kolekciju.

Postupak sortiranja kolekcije u rastućem redoslijedu, ilustrovan je na sl. 4.11. Dijeljenje kolekcije realizuje se veoma jednostavno, ali spajanje uređenih kolekcija zahtijeva pomoćnu kolekciju.

Efikasnost algoritma ne zavisi od inicijalnog rasporeda. **Prosječna složenost** je **linearno-logaritamska**, tj.

$$T(n) = O(n \log n).$$



Slika 4.11: Ilustracija *merge-sort* algoritma

### Implementacija

---

```

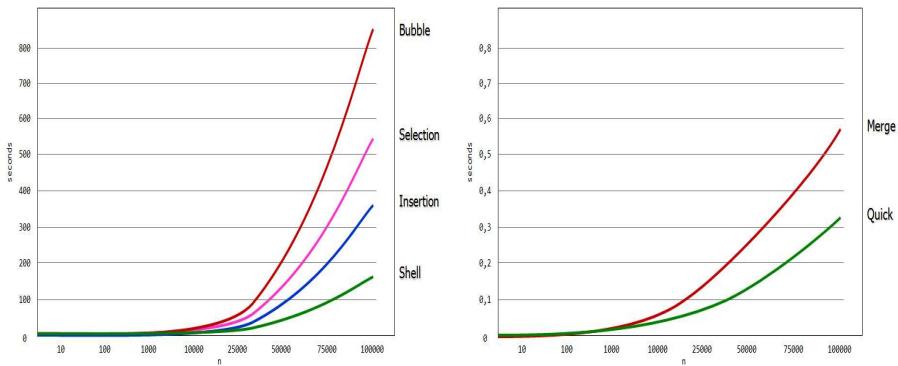
void mergeSort(int niz[], int begin, int end)
{
    if (begin<end)
    {
        /* dijeljenje */
        int sredina = (begin+end)/2;
        mergeSort(niz, begin, sredina);
        mergeSort(niz, sredina+1, end);
        /* spajanje */
        int len = end-begin+1;
        int pom[len]; // pomocni niz
        int i=begin, j=sredina+1, k=0;
        while (i<=sredina && j<=end)
            pom[k++]=niz[i]<=niz[j] ? niz[i++]:niz[j++];
        while (i<=sredina) pom[k++]=niz[i++];
        while (j<=end) pom[k++]=niz[j++];
        for (i=0; i<len; i++) niz[begin+i]=pom[i];
    }
}

```

---

## 4.7. Poređenje metoda zasnovanih na poređenju

Tehnikama koje su zasnovane na poređenju elemenata (ključeva) ne može da se postigne manja vremenska složenost od  $O(n \log n)$  u najgorem slučaju. Za ilustraciju, na sl. 4.12. prikazani su grafici funkcija složenosti za prethodno prikazane algoritme. Zasebno su prikazani algoritmi kvadratne složenosti (*bubble-sort*, *selection-sort*, *insertion-sort*, *shell-sort*), a zasebno algoritmi linearno-logaritamske složenosti (*quick-sort*, *merge-sort*)



Slika 4.12: Grafici funkcija složenosti za algoritme zasnovane na poređenju