



# CS203 - ALGORITMI I STRUKTURE PODATAKA

Sortiranje 1

Lekcija 05

PRIRUČNIK ZA STUDENTE

# CS203 - ALGORITMI I STRUKTURE PODATAKA

## Lekcija 05

### **SORTIRANJE 1**

- ✓ Sortiranje 1
- ✓ Poglavlje 1: Uvod u sortiranje nizova
- ✓ Poglavlje 2: Sortiranje mehurom – bubble sort
- ✓ Poglavlje 3: Metode bazirane na umetanju
- ✓ Poglavlje 4: Sortiranje spajanjem – merge sort
- ✓ Poglavlje 5: Brzo sortiranje – quick sort
- ✓ Poglavlje 6: Vežbe
- ✓ Poglavlje 7: Zadaci za samostalan rad
- ✓ Poglavlje 8: Domaći zadatak
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ❖ Uvod

### UVOD

*Ova lekcija treba da ostvari sledeće ciljeve:*

Sortiranje je proces preuređenja niza elemenata tako da se formira logičan međusobni odnos tih elemenata u preuređenom nizu. Na primer, vaša kreditna kartica prikazuje izvršene transakcije sortirane po datumu — očigledno je da su transakcije preuređene u takav redosled primenom algoritma za sortiranje. Ili, Gmail sortira primljene mejlove po datumu i prema nazivu pošiljaoca – što je još jedan primer primene sortiranja. Sortiranje skupa podataka je preduslov za njegovo efikasno pretraživanje. Podaci koje treba sortirati mogu biti celi brojevi, brojevi dvostrukе dužine, karakteri ili objekti.

Sortiranje je od velikog značaja za izradu velikog broja aplikacija. Izrada i primena najboljeg algoritma u velikoj meri utiče na brzinu rada aplikacija koje operišu sa ogromnim brojem podataka.

**Treba znati da je jedan algoritam sortiranja (quicksort) uvršćen u Top 10 algoritama u 20. veku iz oblasti nauke i inženjerstva.**

U okviru ove lekcije ćemo opisati nekoliko uobičajenih metoda sortiranja, i uporediti njihove performanse. Prvi deo lekcije će biti posvećen opisu nekoliko elementarnih metoda (bubble sort, insertion sort). Neki od razloga zašto uopšte izučavamo ove metode su:

**Prvo**, one nam omogućavaju da usvojimo odgovarajuće termine i da opišemo osnovni mehanizam kako se vrši sortiranje.

**Drugo**, ovi prosti algoritmi su u nekim primenama efikasniji nego sofisticirаниji algoritmi koji su ostavljeni za drugi deo lekcije (mergesort, quicksort).

**Treće**, kao što ćemo imati prilike da vidimo, ovi elementarni algoritmi su korisni za unapređenje sofisticiranih algoritama.

U cilju poređenja algoritama posmatraćemo broj poređenja i razmena podataka pri sortiranju različitih tipova, samim tim ćemo izvršiti analizu vremenske složenosti, i analiziraćemo dodatan memorijski prostor neophodan za izvršenje nekog od algoritma.

# ✓ Poglavlje 1

## Uvod u sortiranje nizova

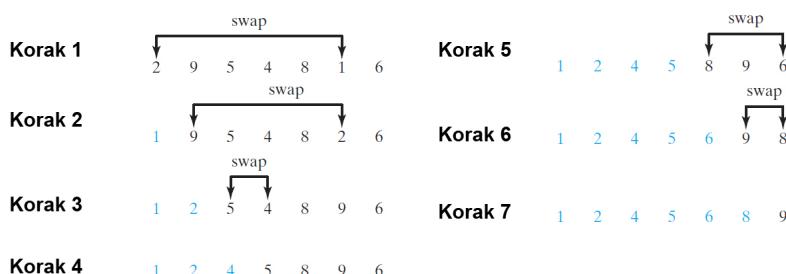
### METODA SELEKCIJE (SELECTION SORT)

*Metoda selekcije pronalazi najmanji element u listi i zamenjuje ga sa prvim elementom. Zatim se u ostatku liste ponovo pronalazi najmanji element i zamenjuje sa drugim članom liste*

Sortiranje, kao i pretraživanje, je veoma čest zadatak u programiranju. Mnogo različitih algoritama je razvijeno u svrhu sortiranja. U nastavku ćemo se upoznati sa jednim veoma prostim i intuitivnim algoritmom a to je metoda selekcije, odnosno *selection sort*.

#### Metoda selekcije - *Selection Sort*

Prepostavimo da želimo da sortiramo listu brojeva u rastućem poretku. Metoda selekcije pronalazi najmanji element u listi i zamenjuje ga sa prvim elementom. Zatim se u ostatku liste, bez prvog člana, ponovo pronalazi najmanji element i zamenjuje sa drugim članom liste. Postupak se ponavlja sve dok se prođe kroz sve pozicije u listi.



Slika 1.1 Postupak sortiranja metodom selekcije [1]

Koraci u izvršavanju algoritma su (Slika 1.1):

**Korak 1:** Izabratи 1 (najmanji) i zameniti ga sa 2 (prvi) u listi.

**Korak 2:** Broj 1 je sada na pravilnoj poziciji i ne treba više ništa razmatrati. Izabratи broj 2 (najmanji) i zameniti mesta sa brojem 9 (prvi) u ostatku liste.

**Korak 3:** Broj 2 je sad postavljen na korektno mesto, i završeno je razmatranje za ovu poziciju. Izabratи 4 (najmanji) i zameniti mu mesta sa 5 (prvi) u ostatku liste.

**Korak 4:** Broj 4 je na pravoj poziciji i njega više ne poredimo sa drugim brojevima. Proveravamo broj 5, ali je on najmanji među elementima u ostatku. Nema nikakve zamene u ovom slučaju.

**Korak 5:** Broj 5 je na korektnoj poziciji, prelazimo stoga na sledeću poziciju. Izaberemo broj 6 (najmanji) i zamenimo mu mesta sa brojem 8 (prvi) u ostatku liste.

**Korak 6:** Broj 6 je sada na korektnoj poziciji i više ga ne uzimamo u razmatranje. Zatim izaberemo broj 8 (najmanji) i zamenimo mu mesta sa 9 (prvi) u ostatku liste.

## IMPLEMENTACIJA SELECTION SORT-A

*Metod selekcije je najjednostavniji, ali i najsporiji metod sortiranja nizova. Sastoje se iz dve ugnezđene petlje koje se maksimalno izvršavaju n puta, gde je n dimenzija niza*

Sada kada znamo kako funkcioniše **selection sort** možemo da ga implementiramo u nekom programskom jeziku (C ili Java na primer). Početnicima u programiranju je teško da iz prvog pokušaja razviju kompletno rešenje problema. Stoga, krenućemo sa pisanjem koda prve iteracije sortiranja sa ciljem da se pronađe najmanji element i da se zameni sa prvim elementom, a zatim ćemo razmisliti šta će biti različito za drugu iteraciju, šta za treću itd. Uvid u svaki pojedinačni korak će vam omogućiti da generalizujete algoritam i da napišete opšti algoritam za bilo koju iteraciju sortiranja. Rešenje može biti predstavljeno na sledeći način:

```
for (int i = 0; i < list.length - 1; i++)  
{  
    pronadji najmanji element iz list[i... list.length-1];  
    zameni (swap) najmanji sa elementom list[i], ako je potrebno;  
    // list[i] je sada na korektnoj poziciji.  
    // U sledecoj iteraciji uradi isto nad list[i+1..list.length-1]  
}
```

U nastavku je listing kompletног programa u Java-i. Tu imamo metod **selectionSort(double[] list)** koji sortira bilo koji niz realnih brojeva tipa **double**. Metod je implementiran korišćenjem ugnezđene **for** petlje.

```
public class SelectionSort {  
    public static void sort(double[] list) {  
        for (int i = 0; i < list.length - 1; i++) {  
            // Pronadji najmanji u list[i..list.length-1]  
            double currentMin = list[i];  
            int currentMinIndex = i;  
  
            for (int j = i + 1; j < list.length; j++) {  
                if (currentMin > list[j]) {  
                    currentMin = list[j];  
                    currentMinIndex = j;  
                }  
            }  
  
            if (currentMinIndex != i) {  
                list[currentMinIndex] = list[i];  
            }  
        }  
    }  
}
```

```

            list[i] = currentMin;
        }
    }
}

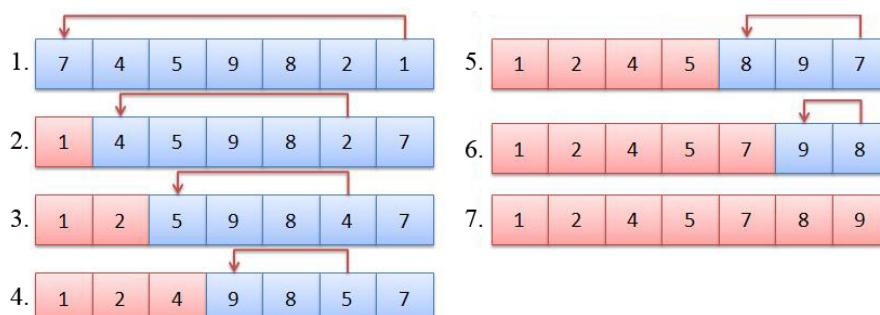
```

Spoljašnja petlja (brojač  $i$ ) iteriše sa ciljem da se pronađe najmanji element u listi koja obuhvata elemente od  $list[i]$  do  $list[list.length-1]$ , a zatim da taj element zameni sa članom  $list[i]$ . Promenljiva  $i$  je inicijalno jednaka  $0$ . Nakon svake iteracije spoljašnje petlje, mi postavljamo odgovarajući član na pravo mesto, a to je mesto  $list[i]$ . Konačno, svi elementi se postavljaju na prava mesta i dobijamo sortiranu listu.

## ANALIZA SELECTION SORT ALGORITMA

*Pošto se sortiranje selekcijom sastoji iz dve ugnezdenе petlje koje se u najgorem slučaju izvršavaju  $n-1$  puta, može se zaključiti da se radi o algoritmu kvadratne složenosti*

Algoritam sortiranja selekcijom pronalazi najmanji element u listi, a zatim menja mesta najmanjem i prvom članu liste. Zatim u ostatku liste (bez prvog člana, koji je sada najmanji) ponovo pronalazi najmanji i menja mu mesta sa prvim članom te preostale liste, i tako dalje dok sa desne strane ne ostane lista sa samo jednim elementom u njoj (Slika 1.2).



Slika 1.2 Postupak sortiranja selekcijom (selection sort algoritam) [1]

Broj poređenja je  $n - 1$  za prvu iteraciju,  $n-2$  za drugu, i tako dalje. Označimo sa  $T(n)$  vremensku složenost **selection sort** algoritma a sa  $c$  ukupan broj ostalih operacija dodele i dodatnog poređenja u svakoj iteraciji. U tom slučaju imamo:

$$\begin{aligned}
 T(n) &= (n - 1) + c + (n - 2) + c + \dots + 2 + c + 1 + c \\
 &= \frac{(n - 1)(n - 1 + 1)}{2} + c(n - 1) = \frac{n^2}{2} - \frac{n}{2} + cn - c \\
 &= O(n^2)
 \end{aligned}$$

Slika 1.3 Vremenska složenost selection sort algoritma [1]

Stoga, ako ostavimo samo najdominantniji član polinoma, složenost **selection sort** algoritma je  $O(n^2)$ .

## ✓ Poglavlje 2

### Sortiranje mehurom – bubble sort

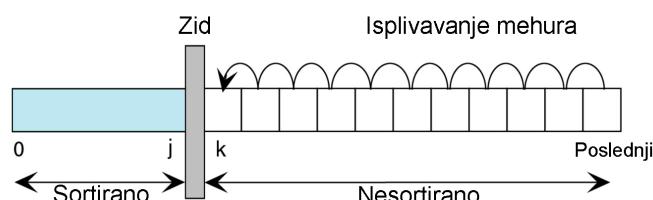
#### OSNOVI O SORTIRANJU MEHUROM

*Sortiranje mehurom ispituje niz od početka do kraja, upoređujući elemente u toku prolaza. Cilj je da se u toku procesa poređenja vrši isplivavanje najlakšeg mehura na jednu stranu niza*

Sortiranje mehurom ispituje niz od početka do kraja upoređujući elemente u toku prolaza. Svaki put kada nađe veći element pre manjeg elementa oni zamenjuju mesta. Na ovaj način se veći elementi prenose prema kraju niza. Najveći (ili najmanji) element “**isplivava**” na jedan kraj niza. Onda se ponavlja proces za nesortirani deo niza sve dok se čitav niz ne sortira.

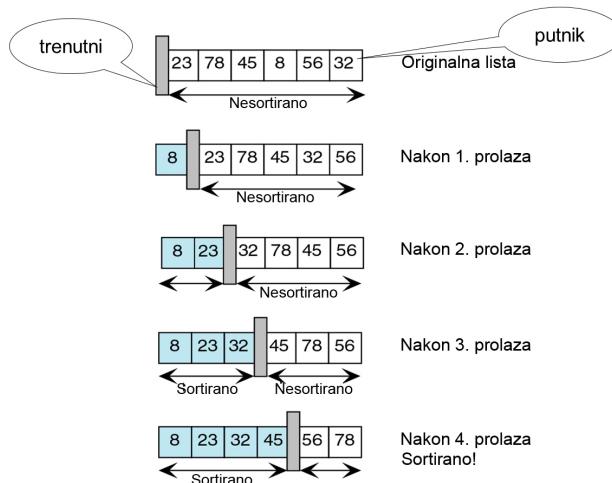
#### Primer sortiranja mehurom

- Lista se deli na dva dela: sortirani i nesortirani deo (Slika 2.1).
- Najmanji element ispliva iz nesortiranog dela u sortirani deo.
- Zid se pomera za jedan element udesno.
- Sortiranje zahteva  $n-1$  prolaza za sortiranje niza.

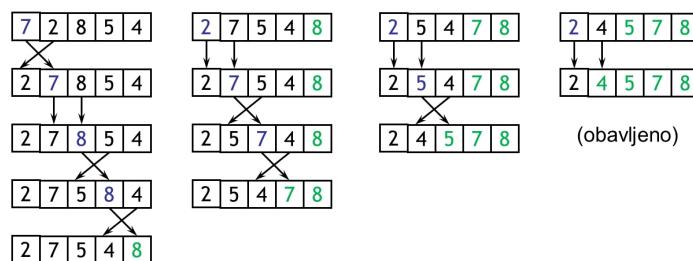


Slika 2.1 Postupak sortiranja mehurom [izvor: autor]

Na Slici 2.2 je primer sortiranja mehurom, pri čemu najmanji element isplivava na početak niza, dok je na Slici 2.3 primer gde najveći element potanja na desnu stranu niza.



Slika 2.2 Primer sortiranja mehurom. Pomeranje zida koje razdvaja sortirani i nesortirani deo [izvor: autor]



Slika 2.3 Sortiranje mehurom. Postupak zamene elemenata na odgovarajućim pozicijama, i pomeranje zida između sortiranog i nesortiranog dela niza [izvor: autor]

## IMPLEMENTACIJA SORTIRANJA MEHUROM

*Svaki put kada najđe veći element pre manjeg elementa, zamenjuju se ova dva elementa. Najlakši ispliva, a proces se ponavlja za nesortirani deo niza sve dok se čitav niz ne sortira*

Algoritam za sortiranje mehurom je prikazan u sledećem listingu

```

for (int k = 1; k < list.length; k++) {
    // Izvrsti k-ti prolaz
    for (int i = 0; i < list.length - k; i++) {
        if (list[i] > list[i + 1])
            swap list[i] with list[i + 1];
    }
}

```

Može se primeti da, ukoliko se ne desi nijedna zamena (swap) u toku jednog prolaza, onda nema potrebe da se uđe u sledeći prolaz pošto su elementi već sortirani. Ovo svojstvo može biti iskorišćeno da se poboljša algoritam, što je prikazano u nastavku:

```
public static void bubbleSort(int[] list) {  
    boolean needNextPass = true;  
  
    for (int k = 1; k < list.length && needNextPass; k++) {  
        // Niz je možda sortiran pa naredni prolaz nije potreban  
        needNextPass = false;  
        for (int i = 0; i < list.length - k; i++) {  
            if (list[i] > list[i + 1]) {  
                // Zameni list[i] sa list[i + 1]  
                int temp = list[i];  
                list[i] = list[i + 1];  
                list[i + 1] = temp;  
  
                needNextPass = true; // Sledeci prolaz je ipak potreban  
            }  
        }  
    }  
}
```

Neka je veličina niza  $n = \text{list.length}$ . Spoljašnja petlja se izvršava  $n-1$  puta (može se reći  $n$ , dovoljno tačno). Svaki put kada se izvrši spoljašnja petlja (**outer loop**), unutrašnja petlja (**inner loop**) se takođe izvrši. Unutrašnja petlja se izvrši  $n-1$  puta u prvom koraku spoljašnje petlje, linearno opadajući do izvršavanja samo jednom u poslednjem koraku.

Dakle, sortiranje mehurom je algoritam vremenske složenosti  $O(n^2)$ .

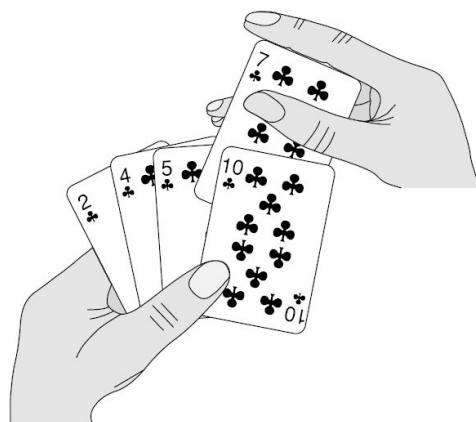
## ▼ Poglavlje 3

# Metode bazirane na umetanju

## IDEJA O SORTIRANJU UMETANJEM

*U okviru metoda sortiranja umetanjem dve najpoznatije metode su: metoda direktnog umetanja (insertion sort) i metoda sortiranja po Šelu (Shell sort)*

Jedan od primera ovakve grupe sortiranja je metod baziran na igri karata gde igrač pokušava da sortira karte u ruci kada uzima novu kartu i pokušava da stavi kartu na pravo mesto (Slika 3. 1).



Slika 3.1 Ideja o umetanju elemenata u niz na određeno mesto [2]

U okviru ove grupe postoje dva tipa sortiranja: Sortiranje po Šelu (autor Donald Shell, 1959) , i Direktno sortiranje umetanjem (insertion sort).

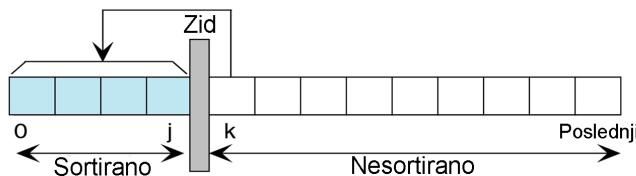
U nastavku će biti obrađeno sortiranje direktnim umetanjem elemenata . U proseku, ova metoda je kvadratne vremenske složenosti i stoga spada u kvadratne algoritme.

## DIREKTNO UMETANJE – INSERTION SORT

*Insertion sort algoritam sortira listu vrednosti tako što uzastopno dodaje novi element u sortiranu podlistu dok cela lista ne postane sortirana*

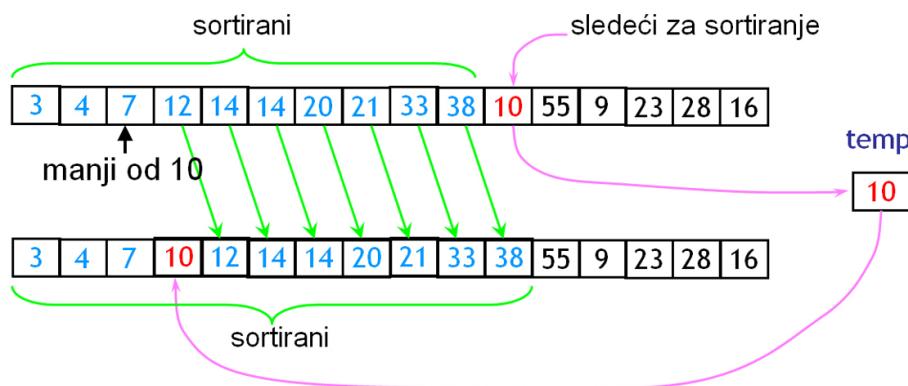
Sortiranje umetanjem je jednostavan algoritam za sortiranje koji generiše krajnji sortirani niz tako što uzastopno dodaje novi element u sortiranu podlistu dok cela lista ne postane sortirana.

Prepostavimo da u datom trenutku u nizu imamo sortirani (levo od zida) i nesortirani podniz, kao što je prikazano na Slici 3.2. U svakom prolazu se prvi element desno od zida, iz nesortirane podliste, prenosi u sortiranu podlistu umetanjem na odgovarajuće mesto.



Slika 3.2 Postupak kod sortiranja direktnim umetanjem [izvor: autor]

U nastavku je prikazan postupak koji se primjenjuje kod sortiranja umetanjem (Slika 3. 3) . Kao što vidimo, imamo dva skupa: sortirani i nesortirani elementi. Sledeći za sortiranje, 10, se pomera u promenljivu temp. Na taj način se omogućava da se svi elementi veći od 10 pomere za jedno mesto u desno, i oslobađa sa prazno mesto na koje se broj 10 iz temp prebacuje u sortirani deo niza.



Slika 3.3 Sortiranje direktnim umetanjem. Postupak pomeranja zida koji razdvaja sortirani od nesortiranog dela [izvor: autor]

Studenti mogu pogledati animaciju sortiranja umetanjem na sledećem web linku: <http://liveexample.pearsoncmg.com/liang/animation/web/InsertionSort.html>. U okviru materijala za vežbe se nalazi primer u Java-i koji kreira jednu ovaku animaciju, što će i biti obrađeno na vežbama.

## IMPLEMENTACIJA INSERTION SORT-A

*U svakom prolazu se prvi element desno od zida, iz nesortirane podliste, prenosi u sortiranu podlistu umetanjem na odgovarajuće mesto.*

Insertion sort algoritam može biti predstavljen na sledeći način:

```
for (int i = 1; i < list.length; i++) {  
    insert list[i] into a sorted sublist list[0..i-1] so that  
    list[0..i] is sorted.  
}
```

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

Da bi ubacili element  $list[i]$  u  $list[0..i-1]$ , neophodno je da sačuvamo element  $list[i]$  u neku privremenu promenljivu, recimo `currentElement`. Zatim vršimo sledeće operacije: pomeramo  $list[i-1]$  u  $list[i]$  ako je  $list[i-1] > currentElement$ , pomerimo  $list[i-2]$  u  $list[i-1]$  ako je  $list[i-2] > currentElement$ , i tako dalje, sve dok je  $list[i-k] \leq currentElement$  ili  $k > i$  (tj, kada smo prošli prvi element sortirane podliste). Konačno, dodelimo vrednost `currentElement` elementu  $list[i-k+1]$ .

Implementacija algoritma može biti izvršena na sledeći način:

```
public static void insertionSort(int[] list) {  
    for (int i = 1; i < list.length; i++) {  
        /** Ubaci list[i] u sortiranu podlistu list[0..i-1] tako da  
         * je list[0..i] sortirano. */  
        int currentElement = list[i];  
        int k;  
        for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {  
            list[k + 1] = list[k];  
        }  
  
        // Insertuj trenutni element u polje list[k + 1]  
        list[k + 1] = currentElement;  
    }  
}
```

## ANALIZA INSERTION SORT-A

*Algoritam se sastoji iz dve ugnježdene petlje. Spoljašnja petlja ide po nesortiranoj grupi elemenata, a unutrašnja petlja ubacuje taj element na odgovarajuće mesto u sortiranom delu*

Kao što možemo da vidimo iz prethodnog koda, u okviru spoljašnje petlje se vrši ubacivanje svakog od  $n$  elemenata. U proseku, postoji  $n/2$  elemenata koji su već sortirani. Unutrašnja petlja pomera pola od ovih elemenata, što znači da ih ima  $n/4$ . Zbog toga je vreme potrebno za sortiranje oko  $n^2/4$ . Zanemarivanjem konstanti dobija se  $O(n^2)$ .

**Dakle, sortiranje umetanjem je vremenske složenosti  $O(n^2)$ .**

## ✓ Poglavlje 4

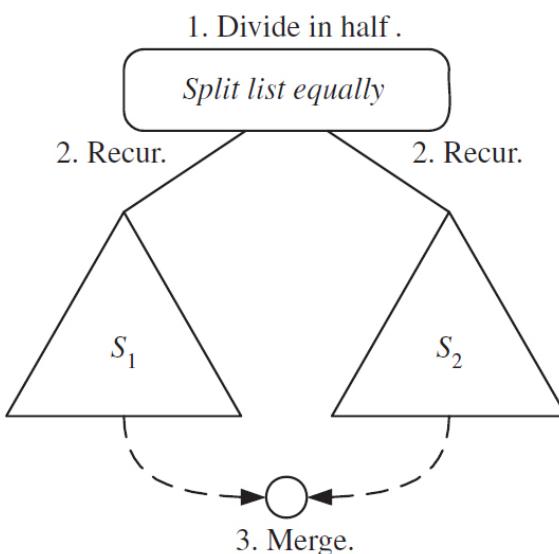
### Sortiranje spajanjem – merge sort

#### OSNOVI SORTIRANJA SPAJANJEM

*Sortiranje spajanjem ili merge sort je rekurzivni algoritam sortiranja zasnovan na poređenju. Predstavlja primer algoritamske strategije podeli i osvoji*

U ovoj sekciji ćemo predstaviti tehniku sortiranja pod nazivom merge sort, ili sortiranje spajanjem, koja može biti opisana na prost i kompaktan način primenom rekurzije.

Predstavlja primer algoritamske paradigmе "podeli i osvoji". Konceptualno, algoritam sortiranja spajanjem se izvršava na sledeći način (Slika 4.1):



Slika 4.1 Podeli i osvoji pristup primjenjen na sortiranje spajanjem [5]

a) Ako niz ima nula ili jedan element, onda je već sortiran.

b) U slučaju da niz ima više elemenata, procedura se sastoji iz sledećih koraka:

- **Divide** (podeli): Podeliti nesortirani niz u dva podniza približno jednake dužine
- **Rekur**: Rekursivno sortirati svaki podniz ponovnom primenom merge sort algoritma.
- **Merge**(spoji): Spojiti dva sortirana podniza u jedan sortirani niz.

Algoritam sortiranja spajanjem je prikazan u sledećem listingu.

```

public static void mergeSort(int[] list) {
    if (list.length > 1) {
        mergeSort(list[0 ... list.length / 2]);
        mergeSort(list[list.length / 2 + 1 ... list.length]);
        merge list[0 ... list.length / 2] with
        list[list.length / 2 + 1 ... list.length];
    }
}

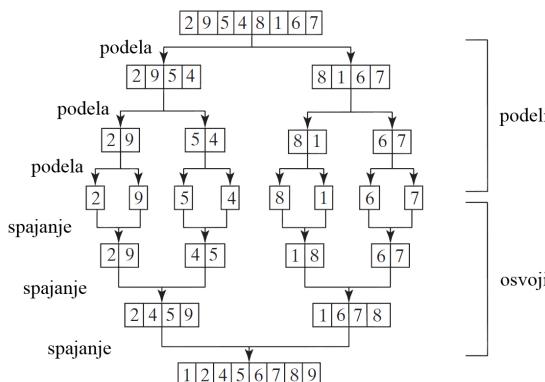
```

U prethodnom kodu `mergeSort` vrši rekurzivno sortiranje jedne od polovina niza (stoga imamo dva poziva `mergeSort`), dok `merge` vrši spajanje dve sortirane polovine u konačan sortirani niz.

## PRIMER SORTIRANJA SPAJANJEM

*Sastoji se iz deljenja nesortiranog niza u dva podniza približno jednake dužine, a zatim nakon rekurzivnog sortiranja svake od polovina vrši se spajanje u uređen niz*

Da bi smo razumeli kako sortiranje spajanjem radi pogledajmo sledeću sliku (Slika 4.2).



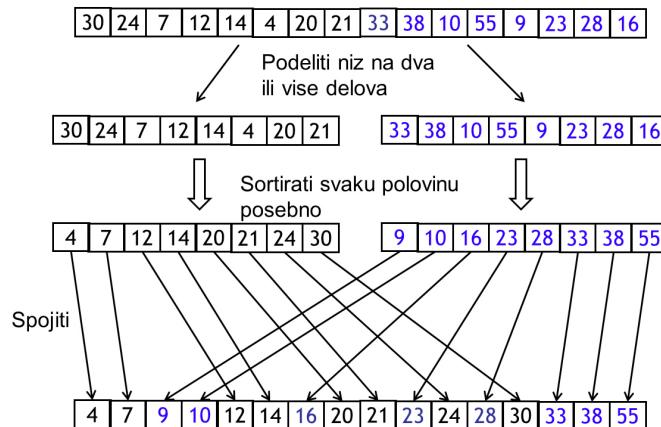
Slika 4.2 Ilustracija sortiranja spajanjem koje koristi "podeli i osvoji" princip [1]

Slika 4.2 ilustruje algoritam spajanjem koji je primenjen na niz od 8 elemenata (2 9 5 4 8 1 6 7). Originalni niz delimo (**divide**) na dva dela (2 9 5 4) i (8 1 6 7). Zatim rekurzivno primenjujemo **merge sort** na svaki od podnizova, sa ciljem da podelimo podniz (2 9 5 4) na dva nova dela (2 9) i (5 4), odnosno drugi podniz (8 1 6 7) na delove (8 1) i (6 7). Ovaj proces se ponavlja sve dok u podnizu ne ostane samo jedan element. Na primer, niz (2 9) delimo na dva podniza (2) i (9).

Pošto sada niz (2) sadrži samo jedan element, on ne može dalje biti podeljen. Sada spojimo (**merge**) podnizove (2) i (9) u novi sortirani podniz (2 9); spojimo (5) i (4) u novi sortirani podniz (4 5). Nakon toga spojimo (2 9) sa (4 5) i dobijemo sortirani podniz (2 4 5 9), i konačno spojimo (2 4 5 9) i (1 6 7 8) u sortirani niz (1 2 4 5 6 7 8 9).

Rekurzivni pozivi nastavljaju da dele niz na manje podnizove, sve dok se u podnizovima ne nađe samo po jedan element. Algoritam zatim spaja ove manje nizove u veće sortirane

podnizove dok se konačno ne dobije jedan sortirani niz. Radi detaljnijeg uvida pogledajmo još jedan primer, Slika 4.3.



Slika 4.3 Sortiranje spajanjem. Postupak podele niza i redosled kako elementi ulaze u spojeni niz [izvor: autor]

## IMPLEMENTACIJA SORTIRANJA SPAJANJEM

*Kod se sastoji iz rekurzivne funkcije mergesort u kojoj se vrši rekurzivna podela niza na polovine (a zatim i na pola od polovine) i funkcije merge koja zatim spaja sortirane polovine*

Program u Javi za sortiranje korišćenjem merge sort-a može biti napisan na sledeći način:

```

public class MergeSort {
    public static void mergeSort(int[] list) {
        if (list.length > 1) {
            // Primeni sortiranje spajanjem na prvu polovinu niza
            int[] firstHalf = new int[list.length / 2];
            System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
            mergeSort(firstHalf);

            // Primeni sortiranje spajanjem na drugu polovinu niza
            int secondHalfLength = list.length - list.length / 2;
            int[] secondHalf = new int[secondHalfLength];
            System.arraycopy(list, list.length / 2,
                secondHalf, 0, secondHalfLength);
            mergeSort(secondHalf);

            // Spoji firstHalf i secondHalf u jednu listu
            merge(firstHalf, secondHalf, list);
        }
    }

    public static void merge(int[] list1, int[] list2, int[] temp) {
        int current1 = 0; // Trenutni indeks u list1
        int current2 = 0; // Trenutni indeks u list2
        int current3 = 0; // Trenutni indeks u temp
    }
}

```

```
while (current1 < list1.length && current2 < list2.length) {  
    if (list1[current1] < list2[current2])  
        temp[current3++] = list1[current1++];  
    else  
        temp[current3++] = list2[current2++];  
}  
  
while (current1 < list1.length)  
temp[current3++] = list1[current1++];  
  
while (current2 < list2.length)  
temp[current3++] = list2[current2++];  
}  
  
public static void main(String[] args) {  
    int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};  
    mergeSort(list);  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
}  
}
```

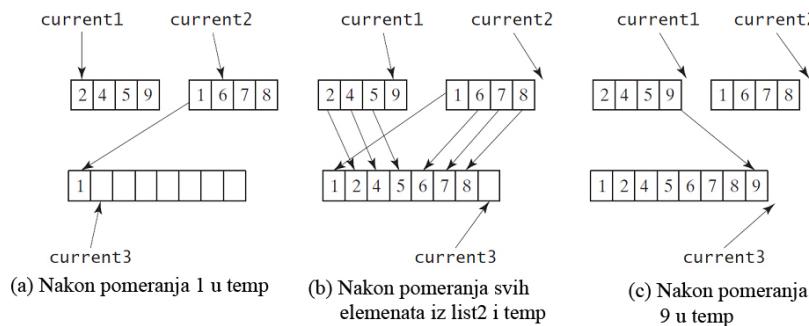
Metod *mergeSort* (linije 3-20) kreira novi niz *firstHalf*, koji predstavlja kopiju prvu polovinu niza *list* (linija 7). Algoritam zatim rekursivno poziva *mergeSort* nad podnizom *firstHalf* (line 8). Dužina niza *firstHalf* je *list.length / 2* dok je dužina niza *secondHalf* jednaka *list.length - list.length / 2*. Drugi niz *secondHalf* se kreira za čuvanje druge polovine originalnog niza *list*. Algoritam poziva *mergeSort* rekursivno i nad podnizom *secondHalf* (linija 15). Nakon što se sortiraju nizovi *firstHalf* i *secondHalf*, vrši se njihovo spajanje u konačan niz *list* (linija 18). Pravilnim spajanjem niz *list* postaje sortiran.

Metod *merge* (linije 23-40) spaja dva sortirana niza *list1* i *list2* u niz *temp*. Indeksi *current1* i *current2* pokazuju na trenutne elemente koji se razmatraju u listama *list1* i *list2* (linije 24-26). Metod uzastopno poređi trenutne elemente nizova *list1* i *list2* i pomera manji od njih u *temp*. *current1* se uvećava za 1 (linija 30) ako je manji onaj iz liste *list1*, odnosno *current2* se uvećava za 1 (linija 32) ako je manji onaj iz liste *list2*. Konačno, svi elementi liste su prebačeni u *temp*. Ukoliko postoje elementi koji su ostali u *list1*, kopiramo ih u *temp* (linije 35-36). U suprotnom, ukoliko ostanu elementi u listi *list2*, njih kopiramo u *temp* (linije 38-39).

## SPAJANJE SORTIRANIH NIZOVA

*Funkcija merge vrši spajanje dva sortirana podniza u jedan konačan sortirani niz*

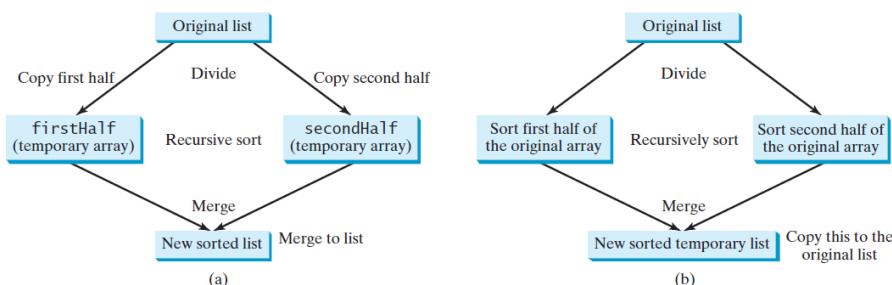
Da bi smo bolje razumeli kako se vrši implementiranje algoritma spajanja pogledajmo Sliku 4.4.



Slika 4.4 Dva sortirana podniza se spajaju u jedan niz [1]

Na Slici 4.4 je prikazano kako se vrši spajanje dva niza *list1* (2 4 5 9) i *list2* (1 6 7 8). Početno, tekući (current) elementi koje razmatramo u nizovima su 2 i 1. Poredimo ih i manji (tj. 1) pomeramo u *temp*, kao što je prikazano na Slici 4.4a. Vrednosti *current2* i *current3* se uvećavaju za 1. Zatim nastavljamo sa poređenjem tekućih elemenata u istim nizovima i opet manji pomeramo u *temp* sve dok se jedan od podnizova potpuno isprazni. Kao što je prikazano na Slici 4.4b, svi elementi niza *list2* su pomereni u *temp* a *current1* pokazuje na element 9 niza *list1*. Konačno, kopiramo 9 u *temp*, kao što je prikazano na Slici 4.4c.

Metod *mergeSort* kreira dva pomoćna niza (linije 6, 12) u toku faze “podeli”, i u pomoćne nizove kopira prvu i drugu polovicu niza (linije 7, 13), sortira pomoćne nizove (linije 8, 15), a zatim ih spaja u originalni niz (linija 18), kao što je prikazano na Slici 4.5a. Možete da izmenite postojeći kod tako da funkcija rekursivno sortira prvu i drugu polovicu niza bez kreiranja novog pomoćnog niza, zatim da spoji dva niza u pomoćni niz, i da zatim sadržaj pomoćnog niza kopira nazad u originalni niz, kao što je prikazano na Slici 4.5b. Ovo možete da uradite kao zadatak za vežbu.



Slika 4.5 Kreiraju se pomoćni nizovi kako bi mogao da se izvrši merge sort [1]

## ANALIZA MERGE SORT-A

*Vremenska složenost algoritma je  $O(n \log n)$ . Ovaj algoritam je stoga bolji od svih kvadratnih algoritama kao što su selection sort, insertion sort, i bubble sort*

Neka  $T(n)$  predstavlja vreme potrebno za sortiranje niza od  $n$  elemenata korišćenjem sortiranja spajanjem. Prepostavimo da je  $n$  stepen broja 2. Merge sort algoritam deli niz na dva podniza, sortira podnizove korištenjem rekurzije i onda spaja dva podniza. Dakle:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + VremeSpajanja(merge)$$

Prvo  $T(n/2)$  je vreme za sortiranje prve polovine niza, a drugo  $T(n/2)$  je vreme za sortiranje druge polovine niza. Za spajanje podnizova je potrebno najviše  $n-1$  poređenja elemenata ova dva podniza i  $n$  pomeranja elemenata u privremenu listu. Ukupno vreme je  $2n-1$ . Imamo da je složenost ovog algoritma:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2n - 1 = O(n \log n)$$

Ovaj algoritam je stoga bolji od svih kvadratnih algoritama kao što su selection sort, insertion sort, i bubble sort.

**Najbolji slučaj:** svi elementi u prvom nizu su manji (ili veći), nego svi elementi u drugom nizu. broj pomeranja:  $2k + 2k$ , a broj poređenja:  $k$ .

**Najgori slučaj:** broj pomeranja:  $2k + 2k$ , broj poređenja:  $2k-1$ .

Merge sort je vremenski jako efikasan algoritam, i najgori i prosečan slučaj su  $O(n \log_2 n)$ . Ali, merge sort zahteva dodatni niz čija je veličina jednaka veličini originalnog niza.

Algoritam sortiranja spajanjem uključuje dva važna principa kojima poboljšava (smanjuje) vremena izvršavanja:

- a. kratki niz je moguće sortirati u manjem broju koraka nego dugački (osnova za deljenje niza na dva podniza)
- b. manje koraka je potrebno za konstrukciju sortiranog niza od dva sortirana podniza nego od dva nesortirana podniza (osnova za spajanje).

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 5

### Brzo sortiranje – quick sort

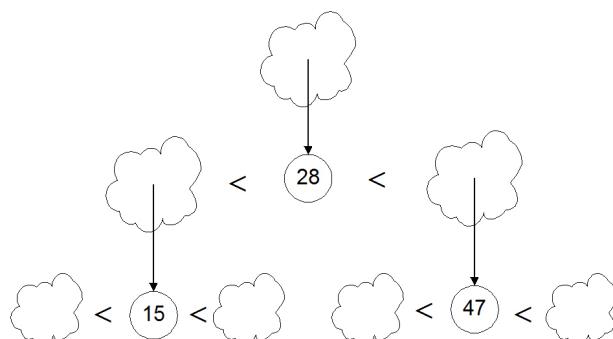
#### UVOD U BRZO SORTIRANJE

*Ideja algoritma sastoji se u podeli niza prema odabranom elementu za podelu koji se dovodi na pravo mesto, i u primeni algoritma brzog sortiranja na svaku od dva dobijena podniza*

Ovo je najčešće korišćen algoritam sortiranja. Nije težak za implementaciju, a koristi manje resursa (vremena i prostora) nego bilo koji drugi algoritam sortiranja, u većini slučajeva.

Ideja algoritma sastoji se u podeli niza prema odabranom elementu za podelu (pivotu) koji se dovodi na pravo mesto, i u primeni algoritma brzog sortiranja na svaku od dva dobijena podniza.

Rekurzivni poziv se završava kada se primeni na podniz sa manje od dva elementa.



Slika 5.1 Postupak brzog sortiranja [5]

Procedura je data na Slici 5.1. Algoritam se sastoji iz sledećih koraka:

- Izabrati "**pivot-a**"
- Podeliti listu na dve liste: jednu sa vrednostima "**manjim ili jednakim**" od vrednosti pivota, i jednu sa vrednostima **većim** od vrednosti pivota
- Sortirati potproblem rekurzivno

Rešenje se dobija spajanjem (engl. **concatenation**) dva podniza.

Pseudokod može biti predstavljen na sledeći način:

```
public static void quickSort(int[] list) {  
    if (list.length > 1) {
```

```

        izaberi pivot
        podeli list na list1 i list2 tako da su
            svi elementi u list1 <= pivot i
            svi elementi u list2 > pivot;
        quickSort(list1);
        quickSort(list2);
    }
}

```

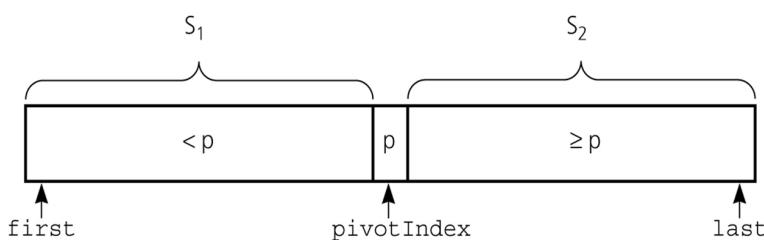
Nakon izvršavanja metode *partition* dovodi se *pivot* na pravo mesto u listi, i deli listu na dve podliste, *list1* i *list2*. Rekursivno, metod *quickSort* se zatim poziva pojedinačno nad podlistama *list1* i *list2*, i radi isti postupak deljenja podlisti na nove podliste prema vrednosti pivota.

## POSTUPAK SORTIRANJA – PODELA I IZBOR PIVOTA

*Prvi korak u fazi sortiranja je izbor pivota. Kao pivot se najčešće bira prvi, srednji ili poslednji element trenutnog podniza*

### Podela

Podela stavlja pivot na pravo mesto u okviru niza (Slika 5. 2).



Slika 5.2 Pravilna podela niza na dva dela koje deli pivot [2]

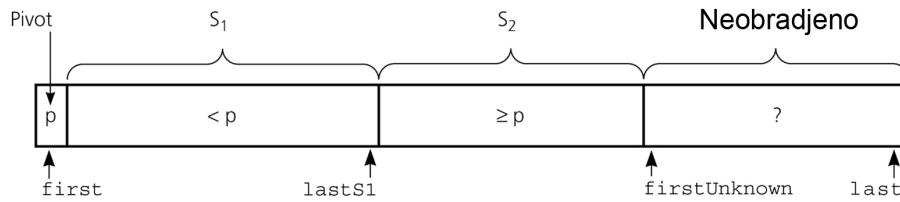
Uređivanje elemenata niza oko pivota *p* generiše dva manja problema pri sortiranju. To su sortiranje levog dela niza i sortiranje desnog dela niza. Kada se ova dva manja problema rešavaju rekursivno, veći problem je time rešen.

### Izbor pivota

Prvo, mora se izabrati pivot element među elementima datog niza, zatim staviti ovaj pivot na prvu lokaciju niza pre podele. *Koji element niza treba da bude izabran kao pivot?*

Ako su elementi niza slučajno odabrani, izabrati pivot nasumice. Može se odabrati prvi ili poslednji element kao pivot, ali možda se tako ne dobije dobra podela. Mogu se koristiti druge metode izbora pivota.

Invarijanta za algoritam podele (Slika 5.3).

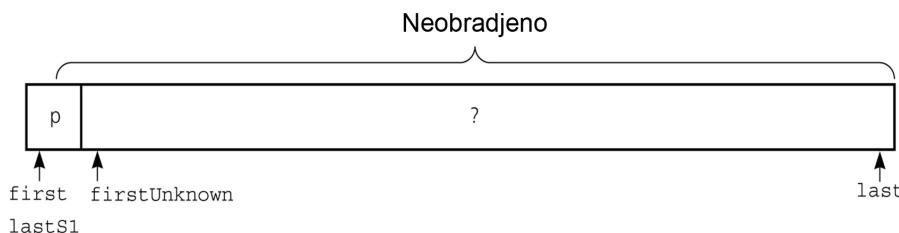


Slika 5.3 Postupak u toku izvršavanja podele niza. Pivot je inicijalno na prvom članu. Koristimo repere lastS1 i firstUnknown da bi smo tačno razdovojili podeljene grupe podnizova [2]

## POSTUPAK SORTIRANJA – ZAMENA PIVOTA

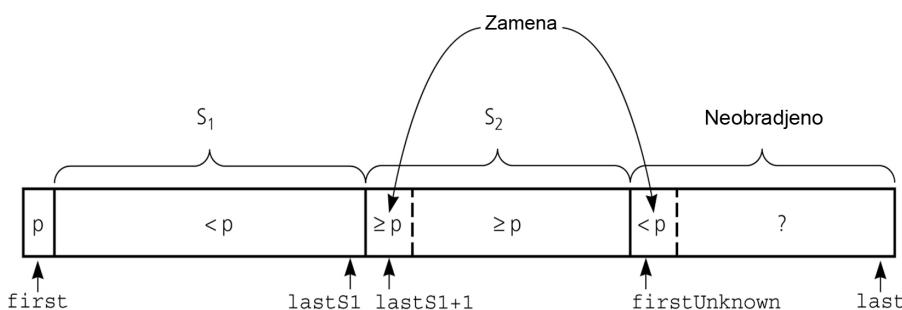
*Nakon što se izvrši podela članova, pivot, koji je inicijalno bio na početku niza, se prebacuje na mesto koje deli elemente koji su manji od njega u odnosu na elemente koji su veći od njega*

Početno stanje (Slika 5. 4) .



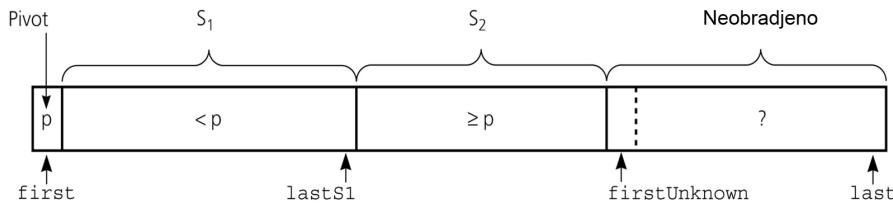
Slika 5.4 Početno stanje podele. Pivot je na prvom članu [2]

Pomeranje `theArray[firstUnknown]` u  $S_1$  zamenom sa `theArray[lastS1+1]` i inkrementiranje  $lastS1$  i  $firstUnknown$  (Slika 5. 5) .



Slika 5.5 Postupak gde element na indeksu  $firstUnknown$  upoređujemo sa pivotom i onda ga pozicioniramo na određeno mesto u obrađenom delu niza [2]

Pomeranje `theArray[firstUnknown]` u  $S_2$  i inkrementiranje  $firstUnknown$  (Slika 5. 6) .



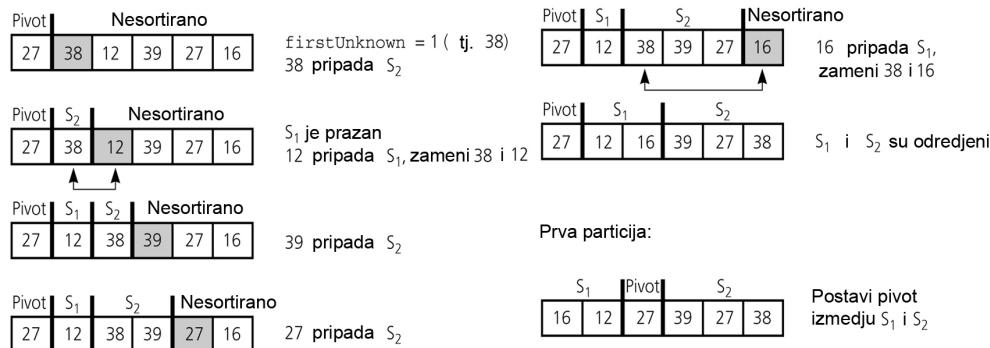
Slika 5.6 Ukoliko je element sa indeksom veći od pivota onda ostaje gde jeste. Indeks firstUnknown zatim uvećavamo za 1 [2]

Ovo je samo jedan od način koji može biti primenjen kada je u pitanju obrada elemenata i postavljanje pivota. U primeru implementacije algoritam ćemo obraditi drugi način rada algoritma, gde algoritam obrađuje elemente istovremeno s leva ne desno, i s desna na levo, tako da se grupa neobrađenih elemenata nalazi u sredini podniza koga trenutno obrađujemo

## PRIMERI BRZOG SORTIRANJA

*Postupak se sastoji iz izbora pivota, podele liste na dva dela prema vrednosti pivota, a zatim na svaku od polovina rekurzivno primeniti identičan postupak*

Slika 5. 7 daje primer brzog sortiranja.



Slika 5.7 Primer brzog sortiranja. Postupak podele niza na dva dela, u zavisnosti od veličine pivota [izvor: autor]

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## IMPLEMENTACIJA BRZOG SORTIRANJA

*Implementacija funkcije quicksort se uglavnom sastoji iz funkcije partition koja pozicionira pivot, i dva rekursivna poziva iste quicksort funkcije koje rade nad polovicama početnog niza*

Quick-sort algoritam u programskom jeziku Java može biti implementiran na sledeći način:

```
public class QuickSort {  
    public static void quickSort(int[] list) {  
        quickSort(list, 0, list.length - 1);  
    }  
  
    private static void quickSort(int[] list, int first, int last) {  
        if (last > first) {  
            int pivotIndex = partition(list, first, last);  
            quickSort(list, first, pivotIndex - 1);  
            quickSort(list, pivotIndex + 1, last);  
        }  
    }  
  
    /** Partition the array list[first..last] */  
    private static int partition(int[] list, int first, int last) {  
        int pivot = list[first]; // Choose the first element as the pivot  
        int low = first + 1; // Index for forward search  
        int high = last; // Index for backward search  
  
        while (high > low) {  
            // Search forward from left  
            while (low <= high && list[low] <= pivot)  
                low++;  
  
            // Search backward from right  
            while (low <= high && list[high] > pivot)  
                high--;  
  
            // Swap two elements in the list  
            if (high > low) {  
                int temp = list[high];  
                list[high] = list[low];  
                list[low] = temp;  
            }  
        }  
  
        while (high > first && list[high] >= pivot)  
            high--;  
  
        // Swap pivot with list[high]  
        if (pivot > list[high]) {  
            list[first] = list[high];  
            list[high] = pivot;  
            return high;  
        }  
        else {  
            return first;  
        }  
    }  
  
    /** A test method */  
    public static void main(String[] args) {  
        int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};  
    }  
}
```

```
quickSort(list);
for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
}
```

Metod *partition* (linije 15–49) deli niz *list[first..last]* primenom pivota na dve particije (dva podniza). Prvi element u prvoj particiji je ustvari *pivot* (linija 16). Početno, *low* pokazuje na drugi element u particiji (line 17) dok *high* pokazuje na poslednji element u particiji (line 18).

Idući s leva u desno kroz niz, metod proverava redom sve elemente i traži prvi koji je veći od pivota (linije 22–23), a zatim pretražuje s desna u levo tražeći prvi element u nizu koji je manji ili jednak od pivota (linije 26–27). Program zatim menja mesta ovim elementima i ponavlja isti proces pretrage i zamene dok se ne obrade svi elementi u okviru *while* petlje (linije 20–35).

Metod konačno vraća novi indeks gde će biti smešten pivot, koji će podeliti trenutni podniz na dva dela ukoliko je pivot u međuvremenu pomeren (linija 44). U suprotnom, metod vraća originalni indeks pivota (linija 47).

## ANALIZA BRZOG SORTIRANJA

*Za najgori slučaj ovaj algoritam je kvadratne složenosti, dok je u najboljem slučaju složenosti  $O(n \log n)$ . U najboljem slučaju ovaj algoritam je bolji i efikasniji od merge sort algoritma*

Da bi se izvršila podela niza od  $n$  elemenata, to zahteva  $n$  poređenja i  $n$  pomeranja u najgorem slučaju. Stoga vreme neophodno za podelu niza je  $O(n)$ .

**Najgori slučaj.** Neka je pivot prvi element. U najgorem slučaju, on svaki put deli niz na dve podliste, pri čemu je jedna od  $n-1$  elemenata, a druga je prazna (Slika-8). Veličina većeg podniza je za jedan manja od veličine niza pre podele:

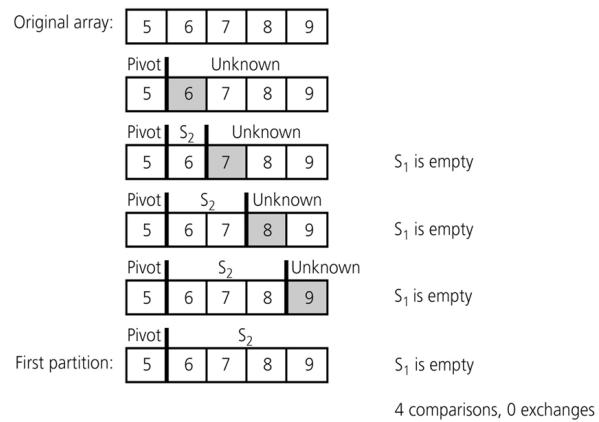
$$T(n)=n-1+n-1+\dots+1=O(n^2).$$

Stoga je quick sort vremenske složenosti  $O(n^2)$  za najgori slučaj.

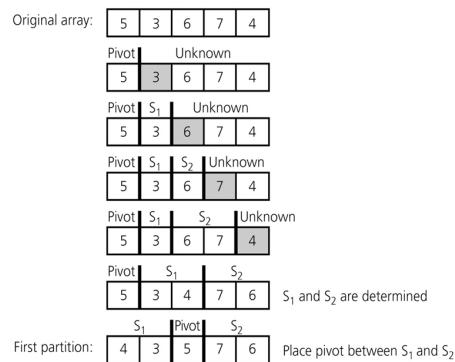
**Najbolji slučaj.** U najboljem slučaju, pivot deli niz na dva podniza približno jednake veličine, oko  $n/2$ . Neka sa  $T(n)$  označimo vreme neophodno za sortiranje niza od  $n$  elemenata korišćenjem brzog sortiranja. Imaćemo:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n = O(\log n)$$

**Prosečan slučaj.** U prosečnom slučaju, pivot neće podeliti niz na dva dela koji su ili približno jednaki ili je jadna potpuno prazna. Statistički, veličine oba podniza su veoma približne. Stoga, vreme izvršavanja prosečnog slučaja je  $O(n \log n)$ , ali matematička analiza prosečnog slučaja nije predmet ove lekcije.



Slika 5.8 Najgori slučaj [2]



Slika 5.9 Najbolji slučaj [2]

## ✓ Poglavlje 6

### Vežbe

#### ZAD 1: GENERIČKI MERGE SORT (20 MIN)

*Cilj zadatka je kreiranje generičkih metoda koje vrše sortiranje korišćenjem metode spajanja*

Napisati sledeća dva generička metoda korišćenjem metode spajanja - merge sort. Prvi metod sortira elemente korišćenjem interfejsa `Comparable` a drugi koristi interfejs `Comparator`.

```
public static <E extends Comparable<E>>
void mergeSort(E[] list)
public static <E> void mergeSort(E[] list,
Comparator<? super E> comparator)
```

Rešenje je dato u nastavku:

```
import java.util.Comparator;

public class Zadatak 1 {
    /** Metod za sortiranje brojeva */
    public static <E extends Comparable<E>> void mergeSort(E[] list) {
        if (list.length > 1) {
            // Primeni sortiranje spajanjem na prvu polovinu liste
            E[] firstHalf = (E[])new Comparable[list.length / 2];
            System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
            mergeSort(firstHalf);

            // Primeni sortiranje spajanjem na drugu polovinu liste
            int secondHalfLength = list.length - list.length / 2;
            E[] secondHalf = (E[])new Comparable[secondHalfLength];
            System.arraycopy(list, list.length / 2,
                            secondHalf, 0, secondHalfLength);
            mergeSort(secondHalf);

            // Spoji firstHalf i secondHalf, tj prvu i drugu polovinu liste
            E[] temp = merge(firstHalf, secondHalf, list);
            System.arraycopy(temp, 0, list, 0, temp.length);
        }
    }

    public static<E extends Comparable<E>> E[]
    merge(E[] list1, E[] list2, E[] temp) {
```

```
int current1 = 0; // Indeks u list1
int current2 = 0; // Indeks u ist2
int current3 = 0; // Indeks u temp

while (current1 < list1.length && current2 < list2.length) {
    if (list1[current1].compareTo(list2[current2]) < 0) {
        temp[current3++] = list1[current1++];
    }
    else {
        temp[current3++] = list2[current2++];
    }
}

while (current1 < list1.length) {
temp[current3++] = list1[current1++];
}

while (current2 < list2.length) {
temp[current3++] = list2[current2++];
}

return temp;
}

public static void main(String[] args) {
    Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
    mergeSort(list);
    for (int i = 0; i < list.length; i++) {
        System.out.print(list[i] + " ");
    }

    System.out.println();
    Circle[] list1 = {new Circle(2), new Circle(3), new Circle(2),
                      new Circle(5), new Circle(6), new Circle(1), new Circle(2),
                      new Circle(3), new Circle(14), new Circle(12)};
    mergeSort(list1, new GeometricObjectComparator());
    for (int i = 0; i < list1.length; i++) {
        System.out.print(list1[i] + " ");
    }

}

public static <E> void mergeSort(E[] list,
    Comparator<? super E> comparator) {
    if (list.length > 1) {
        // Primeni sortiranje spajanjem na prvu polovinu liste
        E[] firstHalf = (E[])new Object[list.length / 2];
        System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
        mergeSort(firstHalf, comparator);

        // Primeni sortiranje spajanjem na drugu polovinu liste
        int secondHalfLength = list.length - list.length / 2;
        E[] secondHalf = (E[])new Object[secondHalfLength];
```

```
System.arraycopy(list, list.length / 2,
                 secondHalf, 0, secondHalfLength);
mergeSort(secondHalf, comparator);

// Spoji firstHalf i secondHalf, tj prvu i drugu polovinu liste
E[] temp = merge(firstHalf, secondHalf, comparator);
System.arraycopy(temp, 0, list, 0, temp.length);
}

private static<E> E[]
merge(E[] list1, E[] list2, Comparator<? super E> comparator) {
    E[] temp = (E[])new Object[list1.length + list2.length];

    int current1 = 0; // Indeks u list1
    int current2 = 0; // Indeks u list2
    int current3 = 0; // Indeks u temp

    while (current1 < list1.length && current2 < list2.length) {
        if (comparator.compare(list1[current1], list2[current2]) < 0) {
            temp[current3++] = list1[current1++];
        }
        else {
            temp[current3++] = list2[current2++];
        }
    }

    while (current1 < list1.length) {
        temp[current3++] = list1[current1++];
    }

    while (current2 < list2.length) {
        temp[current3++] = list2[current2++];
    }

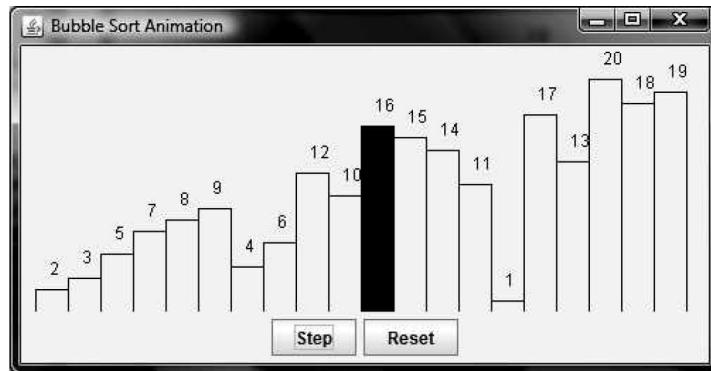
    return temp;
}
}
```

## ZAD 5: ANIMACIJA BUBBLE SORT-A (25 MIN)

*Cilj zadatka je kreiranje programa koji vrši animaciju postupka sortiranja niza primenom metode sortiranja mehurom*

**Zadatak 10.** Napisati JavaFx program koji animira bubble sort. Kreirati niz koji se sastoji od 20 brojeva, nasumično (**random**) raspoređenih u opsegu od 1 do 20. Elementi niza su prikazani u histogramu kao na Slici- 1. Klikom na dugme **Step** se ostvaruje izvršavanje jedne operacije poređenja u algoritmu i vrši se prefarbavanje histograma za tekući korak sortiranja. Obojiti dirku histograma koja se odnosi na element koji se trenutno zamenjuje. Kada se

algoritam završi, prikazati dijalog koji informiše korisnika o kraju sortiranja. Klikom na dugme *Reset* kreira se novi nasumični niz i započinje njegovo sortiranje.



Slika 6.1 Animacija bubble sort algoritma [1]

Rešenje je dato u nastavku:

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class Zadatak10 extends Application {
    double radius = 2;
    @Override // Override the start method in the Application class
    public void start(Stage primaryStage) {
        HistogramPane pane = new HistogramPane();
        pane.setStyle("-fx-border-color: black");

        Button btStep = new Button("Step");
        Button btReset = new Button("Reset");

        HBox hBox = new HBox(5);
        hBox.getChildren().addAll(btStep, btReset);
        hBox.setAlignment(Pos.CENTER);

        BorderPane borderPane = new BorderPane();
        borderPane.setCenter(pane);
        borderPane.setBottom(hBox);

        Label lblStatus = new Label();
        borderPane.setTop(lblStatus);
        BorderPane.setAlignment(lblStatus, Pos.CENTER);
```

```
// Kreiraj scene i postavi ga na stage
Scene scene = new Scene(borderPane, 400, 250);
primaryStage.setTitle("Zadatak 10"); // Postavi naslov stage-a
primaryStage.setScene(scene); // Postavi scenu na stage
primaryStage.show(); // Prikazi stage

StepControl control = new StepControl();
pane.setNumbers(control.getArray());

pane.widthProperty().addListener(ov -> pane.repaint());
pane.heightProperty().addListener(ov -> pane.repaint());

btStep.setOnAction(e -> {
    if (control.step())
        pane.setColoredBarIndex(control.getCurrentIndex());
    else
        lblStatus.setText("The array is already sorted");
});

btReset.setOnAction(e -> {
    control.reset();
    pane.setColoredBarIndex(-1);
    lblStatus.setText("");
});
}

/***
 * Metod main je neophodan za IDE sa ogranicenom JavaFX podrskom
 * Nije potreban za startovanje iz komandne linije
 */
public static void main(String[] args) {
    launch(args);
}

public final static int ARRAY_SIZE = 20;

class StepControl {
    private int[] list = new int[ARRAY_SIZE];

    public int[] getArray() {
        return list;
    }

    StepControl() {
        initializeNumbers();
    }

    public void initializeNumbers() {
        for (int i = 0; i < list.length; i++) {
            list[i] = i + 1;
        }
    }

    // Nasumicna (random) zamena mesta elementima
```

```
for (int i = 0; i < list.length; i++) {
    int index = (int) (Math.random() * ARRAY_SIZE);
    int temp = list[i];
    list[i] = list[index];
    list[index] = temp;
}

private int i = 1;
private int j = 0;

public int getCurrentIndex() {
    return j;
}

public void reset() {
    i = 1;
    j = 0;
    initializeNumbers();
}

public boolean step() {

    if (i >= list.length)
        return false;

    if (j < list.length - i) {
        if (list[j] > list[j + 1]) {
            // Zameni list[i] sa list[i + 1]
            int temp = list[j];
            list[j] = list[j + 1];
            list[j + 1] = temp;
        }
        j++;
    }
    else {
        i++;
        j = 0;
    }

    return true;
}
}

class HistogramPane extends Pane {
    private int[] numbers;
    private int coloredBarIndex = -1;

    public void setNumbers(int[] numbers) {
        this.numbers = numbers;
        repaint();
    }
}
```

```
public void setColoredBarIndex(int index) {
    coloredBarIndex = index;
    repaint();
}

public void repaint() {
    // Pronadji maksimalan ceo broj
    int max = numbers[0];
    for (int i = 1; i < numbers.length; i++) {
        if (max < numbers[i]) {
            max = numbers[i];
        }
    }
}

this.getChildren().clear();

double height = getHeight() * 0.88;
double width = getWidth() - 20;
double unitWidth = width * 1.0 / numbers.length;
for (int i = 0; i < numbers.length; i++) {
    Rectangle bar = new Rectangle(i * unitWidth + 10, getHeight()
        - (numbers[i] * 1.0 / max) * height, unitWidth, (numbers[i] * 1.0 / max)
* height);
    bar.setFill(Color.WHITE);
    bar.setStroke(Color.BLACK);
    this.getChildren().add(bar);
    this.getChildren().add(new Text(i * unitWidth + 10 + 10,
        getHeight() - (numbers[i] * 1.0 / max) * height - 10,
        numbers[i] + ""));
}

if (coloredBarIndex >= 0) {
    int i = coloredBarIndex;
    Rectangle filledRectangle = new Rectangle(i * unitWidth + 10, getHeight()
        - (numbers[i] * 1.0 / max) * height, unitWidth, (numbers[i] * 1.0 / max)
* height);
    filledRectangle.setFill(Color.RED);
    this.getChildren().add(filledRectangle);
}
}
```

## ✓ Poglavlje 7

### Zadaci za samostalan rad

#### ZADACI ZA SAMOSTALNO VEŽBANJE (90 MIN)

*Na osnovu materijala sa predavanja i vežbi pokušati izradu sledećih zadataka:*

**Zad 1 (15 min).** Dve niske su anagrami ako se sastoje od istog broja istih karaktera. Napisati program koji proverava da li su dve niske karaktera anagrami. Niske se zadaju sa standardnog ulaza i neće biti duže od 127 karaktera.

**Uputstvo:** Napisati funkciju koja sortira slova unutar niske karaktera, a zatim za sortirane niske proveriti da li su identične.

**Zad 2 (20 min).** U datom nizu brojeva treba pronaći dva broja koja su na najmanjem rastojanju. Niz se zadaje sa standardnog ulaza, sve do kraja ulaza, ali neće sadržati više od 256 i manje od 2 elemenata. Na izlaz ispisati razliku pronađena dva broja.

**Uputstvo:** Prvo sortirati niz

**Zad 3 (15 min).** (Poboljšani quick sort) Quick sort algoritam koji smo obradili na predavanjima bira prvi element kao pivot. Izmeniti program tako da može da se izabere prvi, srednji i poslednji element kao pivot.

**Zad 4 (20 min).** Napisati program koji pronalazi broj koji se najviše puta pojavljivao u datom nizu. Niz se zadaje sa standardnog ulaza sve do kraja ulaza i neće biti duži od 256 i kraći od jednog elemenata. Uputstvo: Prvo sortirati niz, a zatim naći najdužu sekvencu jednakih elemenata.

**Zad 5 (20 min).** Napisati funkciju koja proverava da li u datom nizu postoje dva elementa čiji zbir je jednak zadatom celom broju. Napisati i program koji testira ovu funkciju. U programu se prvo učitava broj, a zatim i niz. Elementi niza se unose sve do kraja ulaza. Prepostaviti da u niz neće biti uneto više od 256 brojeva. Prvo sortirati niz.

## ✓ Poglavlje 8

### Domaći zadatak

#### DOMAĆI ZADATAK - PRAVILA

##### *Pravila za izradu domaćeg zadatka*

Svaki student dobija od asistenta sopstvenu kombinaciju domaćeg zadatka.

Online studenti bi trebalo mailom da se najave, kada budu želeli da krenu sa radom na predmetu i prikupljanjem predispitnih obaveza.

Odgovarajući NetBeans (Eclipse ili Visual Studio) projekat koji predstavlja rešenje domaćeg zadatka smestiti u folder CS203-DZ05-Ime-Prezime-BrojIndeksa. Zipovani folder CS203-DZ05-Ime-Prezime-BrojIndeksa poslati predmetnom asistentu (lazar.mrkela@metropolitan.ac.rs ) u mejlu sa naslovom (subject) CS203-DZ05, inače se neće računati.

Studenti iz Niša predispitne obaveze predaju asistentu u Nišu (jovana.jovanovic@metropolitan.ac.rs i uros.lazarevic@metropolitan.ac.rs).

Student tradicionalne nastave ima 7 dana, od dana kada je dobio mail sa domaćim zadatkom, da uradi i pošalje rešenje za maksimalan broj poena.

Ukoliko student pošalje domaći nakon tog roka, najviše može da ostvari 50% od maksimalnog broja poena.

Studenti online nastave imaju rok da predaju rešene domaće zadatke 10 dana pre termina ispita u ispitnom roku u kome polažu CS203 Algoritmi i strukture podataka.

Vreme izrade: 2h.

## ▼ Zaključak

### REZIME

*Na osnovu svega obrađenog možemo zaključiti sledeće:*

Sortiranje mehurom ([bubble sort](#)) ispituje niz od početka do kraja, upoređujući elemente u toku prolaza, pri čemu se vrši isplivavanje najlakšeg mehura na jednu stranu niza. Spada u grupu algoritama kvadratne složenosti.

U okviru metoda sortiranja umetanjem opisali smo metodu direktnog umetanja ([insertion sort](#)) koja je kvadratne složenosti.

Sortiranje spajanjem ili [merge sort](#) je rekurzivni algoritam sortiranja zasnovan na poređenju. Predstavlja primer algoritamske strategije "podeli i osvoji". U ovu grupu još spadaju i algoritmi hip sort i quick sort.

Ideja [quick sort](#) algoritma sastoji se u podeli niza prema odabranom elementu za podelu koji se dovodi na pravo mesto, i u primeni algoritma brzog sortiranja na svaku od dva dobijena podniza.

I [quick sort](#) i [merge sort](#) su algoritmi tipa „[podeli i osvoji](#)“ . Za [merge sort](#), najveći deo posla odlazi na spajanje dva niza.

Za [quick sort](#), najveći deo vremena odlazi na podelu niza na dva dela. [merge sort](#) je mnogo efikasniji u najgorem slučaju, dok je [quick sort](#) skoro jednako efikasan u prosečnom slučaju. [merge sort](#) zahteva dodatan prostor, pa je [quick sort](#) efikasniji u najboljem slučaju.

### REFERENCE

#### *Korišćena literatura*

- [1] D. Liang, Introduction to Java Programming, Comprehensive Version, deseto izdanje, Prentice Hall, 2014
- [2] M.A. Weiss, Data Structures and Problem Solving Using Java, treće izdanje, Addison Wesley, 2005.
- [3] Nenad Filipovic, Algoritmi i strukture podataka, Masinski fakultet u Kragujevcu, 2010.
- [4] R. Sedgewick, K.Wayne, Algorithms, 4. Izdanje, Pearson Education, 2011.
- [5] Michael T. Goodrich and Roberto Tamassia, Algorithm Design and Applications, John Wiley, 2015.

