PANEVROPSKI UNIVERZITET APEIRON
FAKULTET INFORMACIONIH TEHNOLOGIJA
BANJA LUKA

Seminarski rad

NEKE IMPLEMENTACIJE ALGORITAMA U PROGRAMSKOM JEZIKU "JAVA" SA OSVRTOM NA OPERACIJE SA NIZOVIMA

Nastavni predmet: ***Algoritmi i strukture podataka***

Predmetni nastavnik/asistent:
Prof. dr Zoran Ž. Avramović
Nedeljko Šikanjić

Student:
Siniša Božić
192-20/RITP

Banja Luka, 2021.

SADRŽAJ

UVOD

U svakodnevnom životu koristimo u većoj ili manjoj mjeri određene procedure kojima rješavamo neke potrebe ili probleme. Svaka ljudska aktivnost sadrži određene sekvencirane i formalizovane naredbe koje se najčešće iskustveno ponavljaju. Na taj način, procedure čine dio naših života i njihovo postojanje je staro koliko i ljudski rod. Naravno, sa širenjem svijesti o potrebi strukturiranja ljudskog znanja te pojavom matematike u antičko doba[1], stvorene su osnove za dalji razvoj algoritama, koji su dobili dalju afirmaciju u srednjem vijeku u Persiji i Iraku kroz radove persijskog matematičara Muhameda ibn Muse al-Karizmija.
U današnje doba, koje možemo slobodno nazvati rastuće-tehnokratskim, algoritmi imaju ogroman značaj kako za pojedinca, tako i za kompletno društvo kao cjelinu. Algoritmi su postali tako moćni da mogu da utiču na ishod izbora, određuju šta ćemo da dobijemo kao rezultat Internet pretrage utičući tako na pristup informacijama, koriste se za predviđanje društvenih trendova, berzanskih kretanja te za kreiranje vještačke inteligencije.

Razloga za izučavanje algoritama i njihove implementacije je zaista puno. U računarstvu, da bi se neko uspješno bavio algoritmima, mora razmišljati interdisciplinarno. To znači da mu treba da budu jasni algoritmi kojima se bavi, te mora da zna kako da ih praktično implementira obzirom da se implementacija tiče konkretnog programskog jezika. Drugim riječima, uspješno razumijevanje i pisanje algoritma neće ništa riješiti ako taj algoritam ne prevedemo u nekom programskom, razvojnom okruženju. To predstavlja samu srž studijskih programa koji se bave računarstvom, odnosno programiranjem kao pod-disciplinom.

U ovom radu će se u osnovnim crtama predstaviti osnove implementacije algoritama u programskom jeziku Java. Iako su algoritmi pojam koji se izdiže iznad odabira programskog jezika u kojem će se implementirati, treba imati u vidu da i najbolja ideja može postati loša ako se ne implementira kako treba. Iako je fokus akademskih institucija na C i C++ jezicima, ovi jezici po mišljenju autora ipak su problematični za početnike imajući u vidu kompleksnost njihove sintakse i neke nedostatke koji ipak nisu u duhu vremena u kome živimo[2]. Kao logičan izbor nameću se Java, ili Python, dva najpopularnija programska jezika današnjice, od kojih je za potrebe ovog rada izabrana Java. Kako je i Java prilično izazovna

1 Euklidov algoritam za određivanje najvećeg zajedničkog djelioca datih brojeva je vjerovatno najstariji algoritam koji je i danas u upotrebi.
2 Pod tim nedostacima se misli o nedostatku objektno-orjentisanog programiranja kada je C u pitanju, te na nedostatak automatizovanog upravljanja memorijom (garbage collection) kod C++.

za početnike, da bi se neki algoritam mogao implementirati uspješno u ovom programskom jeziku, prvo je potrebno imati najosnovniji pregled o karakteristikama jezika, tipovima podataka, najčešće korištenim komandama i strukturama podataka. Kako svaki od jezika zasebno implementira sve ove pojedinačne stavke, vjerovatno je najbolje govoriti o algoritmima u datom programskom jeziku jer se time izbjegavaju mnoge nedoumice koje se mogu pojaviti radi razlika u mnogim detaljima implementacije u programskim jezicima koji mogu iziskivati dodatno vrijeme za pronalaženje eventualnih grešaka i testiranje. Na tržištu je dostupno mnogo stručnih i naučnih radova te specijalizovanih knjiga za problematiku algoritama i struktura podataka u Javi tako da je moguće uspješno rješavati stotine i hiljade konkretnih problema u ovom programskom jeziku.

U radu su objašnjene osnovne strukture podataka uključujući primitivne tipove podataka i nizove, te prikazani najkorišteniji algoritmi za pretragu i sortiranje nizova. Korištena je verzija 15 Jave, te su svi programski kôdovi testirani u razvojnom okruženju IntelliJ IDEA Community Edition 2021.1. Naravno, obzirom na multiplatformnost, Java programe je moguće izvršavati na bilo kojem tipu računara što je još jedan razlog za odabir i učenje ovog programskog jezika. Gdje je god bilo moguće, korišteni su nazivi termina, varijabli i drugih naziva na srpskom jeziku, iako treba imati u vidu da je engleski nativni jezik koji koriste svi programski jezici.

1 UOPŠTENO O ALGORITMIMA I STRUKTURAMA PODATAKA

Algoritmi su preteča računarstva. Međutim sa pojavom i omasovljenjem personalnih računara, a naročito Interneta, algoritme nalazimo svuda. Uopštena definicija algoritma je da je to tako definisana računarska procedura koja uzima neke ulazne podatke, i pod određenim, zadatim kriterijumima ih manipuliše te proizvodi izlazne podatke. Algoritam je ispravan ako u svakom svom izvršavanju proizvodi konzistentne izlazne podatke, kojima se rješava neki problem. Neki od oblasti ljudskog djelovanja gdje su algoritmi najprisutniji su:

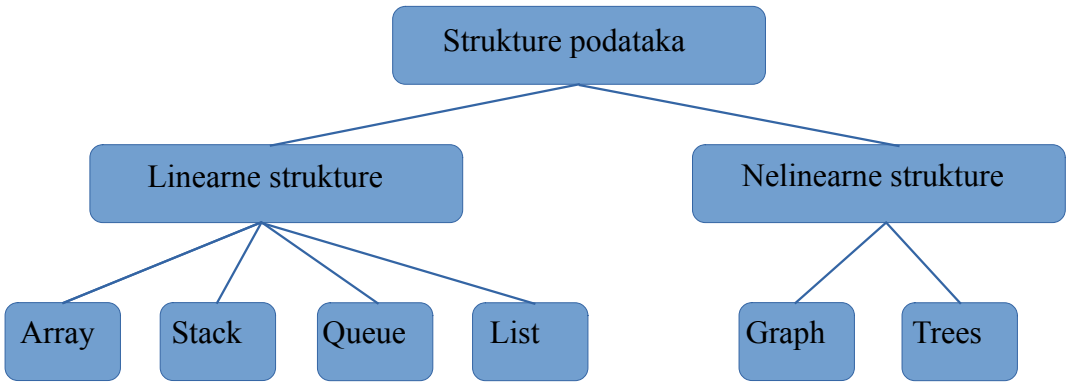
- proizvodnja i logistika, kao i sve srodne djelatnosti gdje je cilj maksimizacija učinka sa datim, ograničenim resursima,
- Internet sajtovi i mrežni resursi koje imaju potrebu za pristupanjem i manipulisanjem velikim bazama podataka,
- e-trgovina, bankarstvo, visokosofisticirane oblasti i istraživanja (genetika, robotika, nanotehnologija),
- vještačka inteligencija i problemi optimizacije.

Kada razmišljamo o kreiranju algoritma za rješavanje nekog problema, ono o čemu trebamo razmišljati je slijedeće:

- šta algoritam rješava (specifikacija problema),
- da li stvarno obavlja željenu funkciju (verifikacija rješenja),
- da li to radi efikasno (analiza performansi).

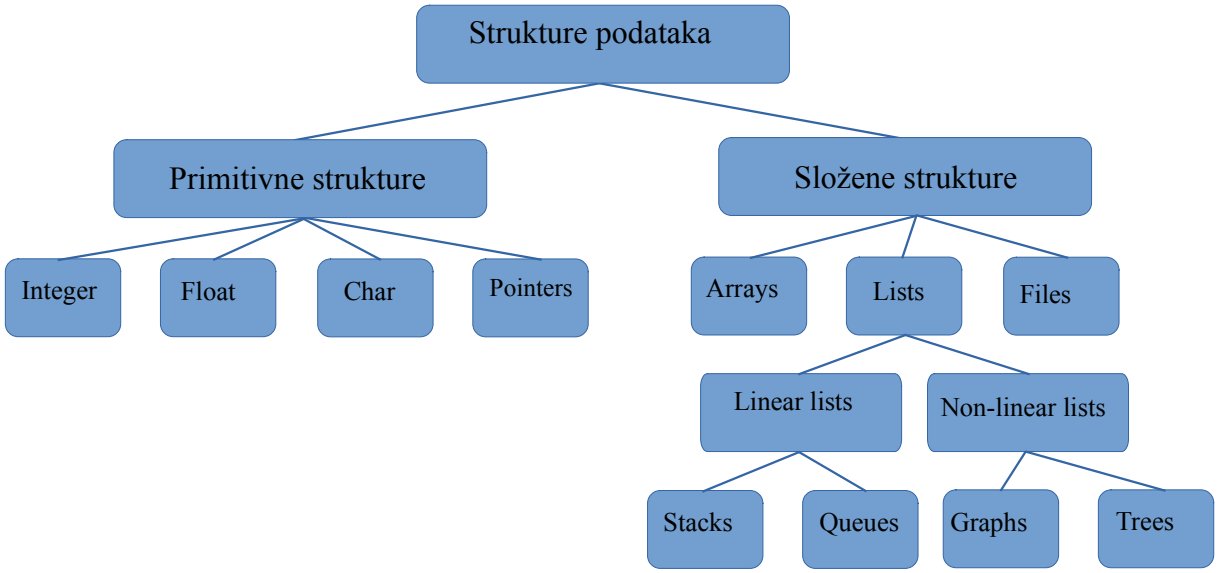
Specifikacija se odnosi na strukturiranje osnovnih premisa koje dati problem predstavlja. Zavisno od problema, nekad će se specifikacija odnositi na detaljan, precizan opis, a nekad će biti više apstraktna. U svakom slučaju će morati da pojasni kako se ulazni podaci trebaju transformisati u izlazne podatke. To može biti jednostavno kod manje složenih problema, međutim kod kompleksnijih, često nije jasno da li algoritam služi svojoj svrsi sve dok ne izvršimo verifikaciju rješenja. Verifikacija rješenja se zapravo odnosi na testiranje postavljenog algoritma, i pretpostavlja da imamo konkretan dokaz pravilnog izvršenja algoritma na datom setu ulaznih podataka. Konačno, analizom performansi se procjenjuje da li predmetni algoritam efikasno koristi ograničene resurse (vrijeme izvršavanja i računarske resurse), odnosno, da li se dati problem može riješiti na bolji i napredniji način.

Strukture podataka se odnose na sve forme organizacije podataka. Jedna od podjela je na linearne i nelinearne:



Prilog 1: Prikaz osnovnih struktura podataka

Još jedna, vrlo često korištena i navođena podjela je na primitivne i složene strukture:



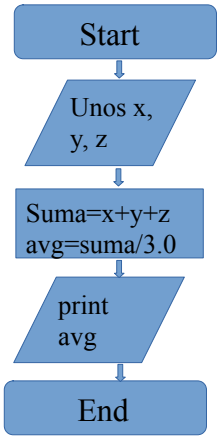
Prilog 2: Prikaz osnovnih struktura podataka

Imajući u vidu nivo apstrakcije, algoritmi se nalaze na sredini između običnog govora kojim se služimo i programskog kôda u datom programskom jeziku. Algoritme ispisujemo u tzv. pseudo-kôdu, koji se sastoji od idućih jezičkih struktura:

- matematičkih, notacionih, logičkih i izraza jednakosti,
- deklaracija funkcija odnosno metoda i njihovih poziva,
- return vrijednosti funkcija/metoda,
- uslovnih izraza (If-Else izraza),

- petlji ("for", "while"),
- indeksiranih vrijednosti nizova i lista (`niz[i], lista(i)`),
- komentara (linija kôda označenih sa `//` ili `/*`, `*/`).

Pseudo-kôd se prvenstveno odnosi na ideje iz viših segmenata apstrakcije, odnosno u ovakvoj notaciji se ne bavimo komandama niskog nivoa (kao su što npr. memorijsko adresiranje, interapti i registri). Glavna prednost korištenja pseudo-kôda je brzina njegovo kreiranja i lakoća razumijevanja datog algoritma. Neki od nedostataka su nedostatak vizuelne reprezentacije[3], nepostojanje unificiranog standarda za izradu pseudo-kôda, kao i potreba njegovim za upravljanjem u smislu ažuriranja i dokumentacije.




```
Begin
unos x
unos y
unos z

suma=x+y+z
avg=suma/3.0

print avg

End
```

Prilog 3: Primjer grafičkog prikaza i pseudo-kôda programa koji računa prosjek tri unesena broja

3 Što se lako rješava korištenjem UML ali zahtjeva dodatno vrijeme i uvećava troškove izrade.

2 VREMENSKA KOMPLEKSNOST ALGORITAMA

Prilikom dizajniranja algoritama, potrebno je imati u vidu ne samo njihovu efektivnost, odnosno da algoritam uspješno rješava dati problem, već i efikasnost, odnosno kako algoritam radi u jedinici vremena. Dakle, vrijeme izvršavanja algoritma je primarno ograničenje[4] koje treba imati u vidu prilikom razvoja datog algoritma, odnosno, govorimo o vremenskoj kompleksnosti. Vremenska kompleksnost bi se mogla procijeniti jednostavnim mjerenjem koliko vremena treba određenom algoritmu za njegovo izvršavanje, ali taj pristup ne može tako lako da se primjeni, odnosno sam po sebi ne bi bio efikasan, jer bi programeri morali da pišu više varijanti programa i da ih mjere, pa potom da najbolji izaberu i koriste u svom programu. To bi višestruko uvećalo vrijeme potrebno za razvoj datog programa a time i troškove. Osim toga to nije ni moguće za velike aplikacije.

Osim toga, treba napraviti razliku između procjene kompleksnosti nekog algoritma, i njegove implementacije. Algoritam je apstraktnija stvar od programskog kôda, može biti pisan u pseudo-kôdu, predstavljen na dijagramu, pa čak i opisan ljudskim govorom. Implementacija algoritma predstavlja stvarni programski kôd pisan u nekom od programskih jezika. Iz tog dalje proizilazi da je mnogo korisnije procjenjivati vremensku kompleksnost algoritama jer se dobija mnogo veća uporedivost, budući da isti algoritam napisan u različitim programskim jezicima gotovo sigurno ima različito vrijeme izvršavanja[5].

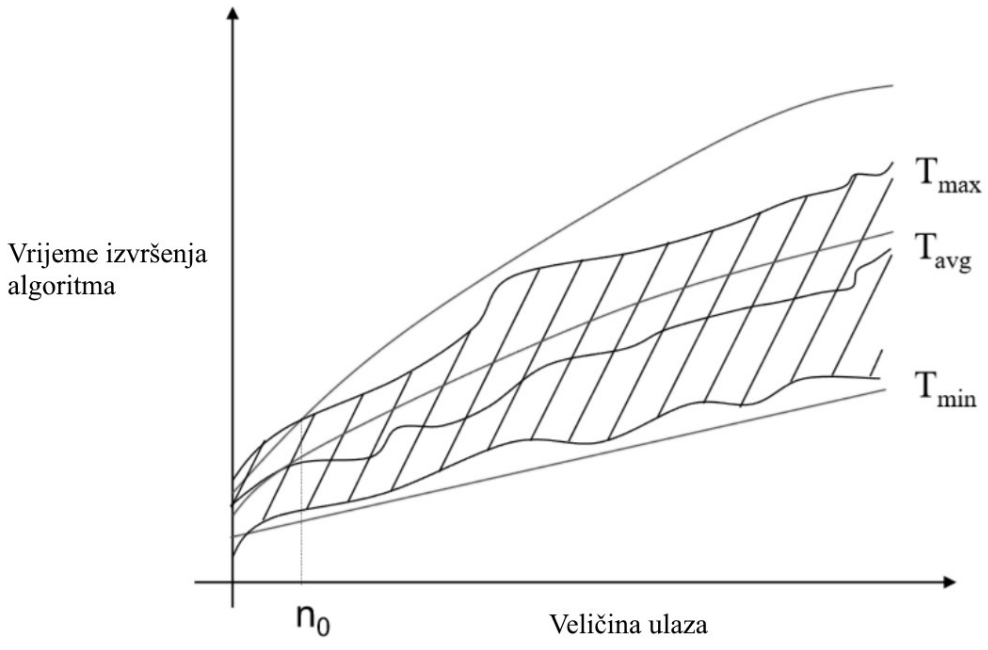
Tip funkcije	Vrijeme izvršavanja
Konstantna	1
Logaritamska	log n
Linearna	n
Kombinovana linearno-logaritamska	n log n
Kvadratna	n^2
Stepenovana	n^m
Subeksponencijalna	$m^{\log n}$
Eksponencijalna	m^n
Faktorijelna	n

Tabela 1: Vremenska kompleksnost nekih uobičajenih funkcija

4 Drugi ograničavajući faktor je zauzeće memorijskog prostora koje će u ovom radu biti zanemareno.

5 Primjera radi, potpuno isti algoritam napisan u programskim jezicima C i Python će biti mnogo brži pri izvršavanju u programskom jeziku C. Ili, program napisan u nekom od asemblerskih programskih jezika će se mnogo brže izvršavati od istog algoritma implementiranog u C.

Vremenska kompleksnost će sigurno zavisiti od veličine problema koji dati algoritam rješava. Na primjer, kod algoritma koji vrši pretragu, kompleksnost će zavisiti od broja elemenata koje pretražujemo, kod sortirajućeg algoritma broj elemenata koji se sortiraju direktno utiče na vrijeme izvršavanja, i tome slično. Shodno tome, kompleksnost algoritma će biti funkcija koja procjenjuje vrijeme izvršavanja nad nekim setom podataka. Naravno, više od 50 godina razvoja računarske tehnike su znatno ubrzale njihove performanse, međutim tema je i dalje aktuelna, jer su ubrzanja ipak manja od razlika u setovima podataka kojima se manipuliše[6]. Drugim riječima, neefikasni algoritmi i dalje predstavljaju ograničavajući faktor u modernom računarstvu te im se treba pristupati sa stanovišta efikasnosti. Vrijeme izvršavanja datog algoritma se može procjenjivati a priori i a posteriori analizom. A priori analizom se koriste matematičke metode za analiziranje vremena izvršavanja algoritma i to prije njegovog pokretanja, dok se a posteriori analiza odnosi na analizu vremena nakon stvarnog izvršavanja algoritma. Na osnovu naprijed rečenog, vremenska složenost algoritma se u smislu a priori analize, kojom ćemo se prevashodno baviti, kao asimptotska složenost funkcije koja predstavlja broj izvršavanja algoritma, imajući u vidu količinu ulaznih podataka. Tako se razlikuju tri slučaja, najgori, najbolji i prosječan slučaj složenosti.



Prilog 4: Prikaz tri slučaja vremenske kompleksnosti

6 Iako je poređenje performanski procesora tema za sebe, možemo tvrditi da su se performanse povećale stotinama i hiljadama puta od predstavljanja prve generacije Intelovih procesora 1970ih, dok su setovi podataka (npr. baze podataka) rasle milijardama i trilionima puta.

Najgori slučaj izvršavanja je vrijeme izvršavanja datog algoritma koje je maksimalno moguće i nikad ne može biti prekoračeno. To je dakle granična vrijednost koja pokazuje kako dati algoritam funkcioniše u najgorem slučaju. Obzirom da su ovakvi algoritmi česti (npr. vrijeme traženja nepostojećeg podatka), najgori slučaj može pružiti uvid u to kako se neki algoritam ponaša te pružiti mogućnost poređenja sa drugim algoritmima. Prosječna složenost je statistička složenost koja uprosječuje ukupno moguće vrijeme izvršavanja sa setom ulaznih podataka. Obzirom da algoritmi mogu biti polinomijalni, nepolinomijalni, subeksponencijalni i eksponencijalni, njihova vremenska složenost raste ovim redom. Tako se algoritmima dodjeljuju oznake iz asimptotske notacije, koja obuhvata:

- O-notaciju (gornja granica),
- Ω- notaciju,
- Θ-notaciju, i
- o-notaciju.

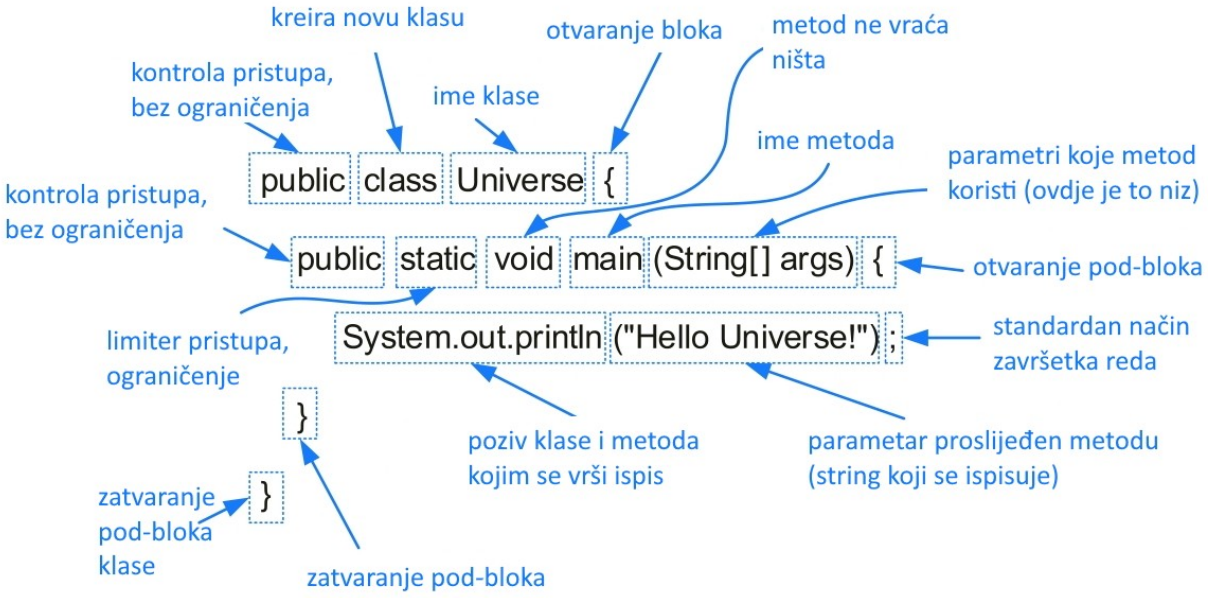
O-notacija je jedna od najčešće korištenih i svodi se na matematičko poređenje dvije nenegativne funkcije `f` i `g`, za koje se može tvrditi da je `f(n)=O(g(n))` ako postoje pozitivne konstante `c` i n_0 takve da za svako $n \geq n_0$ važi `f(n)<=c g(n)`[7].

7 Za detalje pogledati "Introduction to Algorithms", strana 43.

3 OSNOVE IMPLEMENTACIJE ALGORITAMA U PROGRAMSKOM JEZIKU JAVA

Da bi se algoritmi i strukture podataka mogle uspješno implementirati u nekom programskom okruženju, potrebno je poznavati osnove nekog od programskih jezika. Kako smo već naglasili, algoritmi su apstraktna kategorija i eventualno se moraju implementirati u nekom konkretnom programskom jeziku. Ovdje je odabrana Java zbog niza prednosti koje pruža nad nekim drugim konkurentskim rješenjima[8].

Uzmimo za primjer najjednostavniji program koji ispisuje dvije riječi, "Hello Universe". Java je takav programski jezik, sastavljen u potpunosti od klasa i podklasa, u skladu sa načelima objektno-orjentisanog pristupa. To znači da se programskom kôdu pristupa sa stanovišta objekata koji su instance ili već ugrađenih klasa ili klasa koje korisnik stvori.



Prilog 5: Struktura jednostavnog programa u Javi

Sintaksa Jave je djelimično preuzeta iz C/C++ kao i pravila pisanja kôda, odvajanja blokova i redova. U konkretnom primjeru, pošto svaki program u Javi mora biti sadržan u klasi, definišemo klasu `Universe`, i jedan metod[9] u njoj, `main` metod. U tom metodu pozivamo klasu System i njen pod-metod kojim se vrši ispis datog parametra. Linije kôda su odvojene znakom `;` , a segmenti kôda, odnosno blokovi vitičastim zagradama. Svaki blokovi kôda

8 Na primjer, za razliku od C/C++, programer je oslobođen radnji kao što su vođenje računa o životu varijabli, obzirom da Java ima garbage collector, zatim, opet za razliku od navedenih jezika, program pisan u Javi je nezavisan od platforme na kojoj se izvršava i konačno, zato što je Java konstantno prvi ili jedan od top 3 najpopularnijih programskih jezika u zadnjih 20 godina.

9 U C/C++ je to funkcija.

unutar klase imaju određena pravila pristupa, te u principu važi da deklarisana varijabla unutar jednog bloka kôda nije vidljiva u drugim blokovima kôda, a za izuzetak od ovog pravila se koriste modifikatori pristupa, od kojih su najvažniji `public` (koji označava dati blok otvorenim), te `private` (koji zatvara blok kôda onemogućavajući mu pristup izvana). Pored modifikatora pristupa, postoje i modifikatori koji se ne kategorišu u tom smislu[10] i od kojih je najvažniji `static`. Modifikator `static` se koristi da bi se vršila kontrola nad tokom izvršavanja programa i može se koristiti za klase, metode i varijable. Na primjer, glavni metod `main` koji mora postojati u svakom Java programu je `static`, što znači da mu je moguće pristupati bez pozivanja klase u kojoj se nalazi.

Varijable, metodi i klase se mogu proizvoljno imenovati, sa izuzetkom rezervisanih riječi koje ne možemo koristiti obzirom da ih koristi sam programski jezik.

abstract	default	goto	package	this
assert	do	if	private	thwor
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	null	synchronized	

Tabela 2: rezervisane riječi u Javi

Za osnovno razumijevanje Java sintakse potrebno je poznavati tipove podataka, koji se mogu podijeliti na osnovne, takođe poznate kao i primitivne, te složene, odnosno objekte. Osnovni tipovi podataka su:

boolean	logička vrijednost, istinito ili neistinito
char	16-bitni Unicode karakter
byte	8-bitni broj
short	16-bitni broj
int	32-bitni broj
long	64-bitni broj
float	32-bitni broj sa decimalnim mjestom
double	64-bitni broj sa decimalnim mjestom

Tabela 3: osnovni tipovi u Javi

Za kraj ovog kratkog pregleda, komentari se mogu ostavljati u samom kôdu pomoću dvije kose crte, `//`, što označava jednolinijski komentar, ili putem blok-višelinijskog komentara `/*...*/`.

10 Tačnije, po zvaničnoj kategorizaciji, postoje access modificators i non-access modificators.

4 PETLJE

4.1. "FOR" PETLJA

Sintaksni oblik "for" petlje u Java programskom jeziku je[11]:

```
for (inicijalizacija; uslov; iteracija)
{
neki kôd
}
```

Prilikom definisanja ove petlje, potrebno je navesti i inicijalizovati varijablu koja će kontrolisati petlju, odnosno služiti kao njen brojač. Uslov petlje je logički operator koji određuje da li će se petlja ponavljati. Iteracija definiše kako će brojač da se mijenja. Ova petlja će se izvršavati sve dok je uslov `istinit`, i tek kad on promijeni stanje u `nestinit` petlja prestaje i program se izvršava daljim redom odozgo na dole, s lijeva na desno.

Primjer:

```
//ispisuje kvadratni korijen brojeva od 0-99
class KvadratniKorijen {
public static void main(String args[]) {
        int broj;
        double korijen;

        for (broj=1; broj<100; broj++) {
            korijen=Math.sqrt(broj);
                System.out.println("Kvadratni korijen broja "+
broj + " je "+korijen);
          }
     }
}
```

Napomena: definisana je varijabla tipa `int` i varijabla tipa `double` (budući da je korijen decimalni broj). Petlja počinje iteraciju od broja 1 zaključno sa brojem 99, za svaki broj računa kvadratni korijen (pozivajući klasu `Math` i njen metod `sqrt()`), te se vraća na početak, povećava brojač za jedan (izraz `broj++` znači *post* povećanje varijable broj za jedan[12]). Kada dosegne broj 99 petlja završava.

"For" petlja ima više varijacija i mogućih upotreba.

Na primjer, moguće je da se petlja kreće u pozitivnom ili negativnom smjeru, sa proizvoljno velikom iteracijom. Slijedeći program ispisuje brojeve od 100 do -95 u koracima od po 5:

11 Za detaljan opis pogledati Oracle Java Documentation: https://docs.oracle.com/javase/specs/jls/se15/html/jls-14.html#jls-14.14.1

12 Npr. izraz ++broj prvo uvećava broj za jedan pa ga onda ispisuje (bez obzira na neku drugu operaciju koju treba izvršiti sa tim brojem), a izraz broj++ prvo uzima u obzir vrijednost varijable pa je onda povećava.

```
// "for" petlja koja se krece u negativnom smjeru
class negativnaPetlja {
public static void main(String[] args) {
        int broj;
        for (broj=100; broj>-100; broj-=5)
            System.out.println(broj);
     }
}
```

Napomena: kada se radi o jednostavnim izrazima, vitičaste zagrade bloka petlje se mogu izostaviti. Ova petlja počinje od broja 100, uz uslov da je broj veći od -100, te iteriše u dekrementalnim koracima od po 5. Ispisuju se brojevi od 100 do -95 u koracima od 5. Izraz `broj-=5` skraćeno znači: `broj=broj-5`.

"For" petlja testira uslov pri početku, ukoliko bi se desilo da je uslov `neistinit`, petlja se ne bi izvršila i kompajler bi prijavio grešku, kao na slijedećem primjeru:

```
for (broj=10; broj<5; broj++) {
neki kôd
}
```

Ova petlja se uopšte ne bi izvršila jer je početni uslov netačan, odnosno broj 10 nije manji od 5.

"For" petlja omogućava korištenje više brojača i kontrolnih varijabli. Na primjer:

```
// koristenje vise varijabli u "for" petlji
class viseVarijabli {
public static void main(String args[]) {
    int a,b;

    for (a=0, b=10; a<b; a++, b--)
        System.out.println("a i b: " + a + " " + b);
     }
}
```

Napomena: ovo je specifičan i rjeđe korišten slučaj, u kome petlju kontrolišu dve varijable (moguće je i više od toga ali takve petlje postaju komplikovane za razumjevanje). Budući da petlja "for" omogućava inicijalizaciju varijabli, moguće je bilo izbaciti red `int a,b;` :

```
for (int a=0, b=10; a<b; a++, b--)
```

Petlja postavlja inicijalne vrijednosti varijabli `a` i `b`, respektivno, na 0 i 10, postavlja uslov da je `a` manje od `b`, te potom iterira uvećavajući za jedan varijablu `a` te istovremeno smanjujući za jedan varijablu `b`. Program iterira sve dok je `a` manje od `b` te ispisuje:

```
a i b: 0 10
```

```
a i b: 1 9
a i b: 2 8
a i b: 3 7
a i b: 4 6
```

4.2. "WHILE" PETLJA

"While" petlja se u programskom jeziku Java deklariše kao[13]:

```
while (uslov)
{
neki kôd;
}
```

Pri tome, uslov je logički iskaz koji može biti `tačan` ili `netačan`. Dok je uslov `tačan`, petlja se izvršava, kad uslov postane `netačan`, petlja prestaje.

Idući primjer pokazuje kako se jednostavno mogu ispisati slova abecede korištenjem ove petlje.

```
// primjer "while" petlje
class WhilePrimjer {
public static void main(String args[]) {
    char karakter;
    karakter= 'a';
        while (karakter<'z') {
            System.out.print(karakter);
            karakter++;
          }
     }
}
```

Napomena:

Definisana je varijabla tipa `char` (koja se inicijalizuje jednostrukim navodnim znacima). Petlja je tako inicijalizovana da je varijabla karakter, početnog stanja `'a'`, manja od krajnjeg karaktera abecede, odnosno `'z'`. Sve dok je taj uslov tačan, petlja vrši iteraciju i ispisuje znak po znak. Nakon prve iteracije, varijabla karakter se uvećava za jedan, odnosno sa karaktera `'a'` se povećava na `'b'` i tako redom.

Suštinski, petlje "for" i "while" predstavljaju jednu te istu stvar iskazanu na različite načine. Petlja "for" omogućava i zahtjeva deklarisanje i inicijalizaciju varijable tokom deklarisanja same petlje, dok kod "while" petlje se deklarisanje i inicijalizacija kontrolne varijable obavlja prije same petlje. Obadvije petlje koriste logički iskaz za verifikaciju iteracije, dok se brojač u

13 Za detaljan opis pogledati Oracle Java Documentation: https://docs.oracle.com/javase/specs/jls/se15/html/jls-14.html#jls-14.12

"for" petlji također obično deklariše tokom deklarisanja same petlje[14], dok se u "while" petlji brojač deklariše na kraju izvršenja kôda unutar bloka petlje.

Pod-varijanta "while" petlje je "do-while" petlja, čija je osobenost da će se izvršiti barem jednom, bez obzira na istinitost logičkog uslova, i ova petlja se obično rjeđe koristi.

Sumarizovano dajemo pregled ove tri petlje:

Poređenje	**For petlja**	**While petlja**	**Do while petlja**
osnovno	kontrola toka programa koja iteriše blok programa na osnovu logičkog iskaza	kontrola toka programa koja iteriše blok programa na osnovu logičkog iskaza	Kontrola toka programa koja izvršava dio programa bar jednom, bez obzira na stanje logičkog iskaza, a dalji dok izvršavanja (iteracije) zavisi od logičkog iskaza
korištenje	Ako je broj iteracija poznat	Kada je broj iteracija nepoznat ili nije fiksno zadat	Ako se ne zna tačan broj iteracija a postoji potreba da se dio programa izvrši bar jednom
sintaksa	`For(inicijalizacija; uslov; iteracija) { neki kôd }`	`While (uslov) { neki kôd }`	`Do { neki kôd } while (uslov);`
primjer	`For (int i=1; i<=10; i++) { System.out.println(i); }`	`Int i=1; while (i<=10) { System.out.println(i); i++; }`	`Int i=1; do { System.out.println(i); i++; while (i<=10) { }`

Tabela 4: Sumarni pregled karakteristika petlji u programskom jeziku Java

14 "for" petlja se može definisati i kao:
for (int i=0; i <10) {
neki kôd;
i++
}

4.3 BREAK, CONTINUE KAO NAČINI KONTROLE IZVRŠAVANJA PETLJE

Izraz break u bloku petlje prekida njeno izvršavanje, bez obzira na stanje logičkog uslova, i kontrolu izvršavanja programa usmjerava na kôd koji se nalazi nakon petlje. Na primjer:

```
// koristenje break za prekid petlje
class Break {
public static void main(String args[]) {
        for (int i = 0; i < 10; i++){
            if (i == 5)
                break;
            System.out.println("i: " + i);
          }
System.out.println("Petlja zavrsena");
     }
}
```

Napomena:

Program ispisuje brojeve od 0-9. Ova petlja ispisuje brojeve počev od 0, iteracija povećava korak po korak vrijednost brojača. Kada brojač bude imao vrijednost 5, `break` prekida petlju. Program ispisuje vrijednosti varijable i od 0 – 4. Ukoliko bi umjesto `break` unijeli `continue`, program bi nastavio ispisivanje sve do broja 9. Dakle, za razliku od `break`, `continue` ne izlazi iz petlje već nastavlja njenu iteraciju sve do momenta kada logički uslov sa početka petlje postane neistinit.

5 NIZOVI (ARRAYS)

Niz predstavlja skup varijabli istog tipa, kojima je dodjeljeno isto ime. U programskom jeziku Java, nizovi mogu biti višedimenzionalni[15], ali se najčešće koriste jednodimenzionalni. Nizovi rješavaju problem dodjele više vrijednosti jednoj varijabli. Npr. niz može da sadrži spisak isplaćenih plata, listu dnevnih temperatura, dobitaka na lutriji itd. U Javi, niz predstavlja objekat (za razliku od običnih, tzv. primitivnih varijabli), što pravi određenu razliku u manipulaciji nizovima a naprednijim korisnicima može donijeti i određene benefite[16].

Opšta sintaksa za kreiranje niza je[17]:

```
tip ime niza [] = new tip[velicina]
```

Pošto su nizovi objekti, prilikom kreiranja niza prvo se definiše tip niza i naziv varijable niza, a onda (kao i svi drugi objekti), koristi se `new` čime se alocira memorija. Na primjer, idući niz kreira niz brojeva sa 10 elemenata i dodjeljuje ih varijabli `primjer`:

```
int primjer[] = new int[10];
```

Niz deklarisan na ovaj način je stvoren kao prazan objekat, te ga je moguće popuniti iteriranjem i dodjelom pojedinačnih vrijednosti njegovim indeksnim brojevima:

```
// deklarisanje, inicijalizacija, popunjavanje i ispis
jednodimenzionalnog niza
class Niz {
public static void main(String args[]) {
int primjer[]=new int[10];

          for (int i = 0; i < 10; i++)
            primjer[i]=i;

          for (int i = 0; i < 10; i++)
               System.out.println("Element   ["   +i   +   "]:
"+primjer[i]);
     }
}
```

Napomena: iz navedenog primjera se vidi da se niz popunjava iteracijom, kao da se iteracijom i ispisuje njegov sadržaj.
Indeksi niza počinju od 0, dakle ako niz ima veličinu 10, indeksi se kreću od 0-9. To je tako u većini programskih jezika pa i u Javi.
Niz je moguće popuniti i odmah, pri deklarisanju i inicijalizaciji:

15 Npr. dvodimenzionalni niz je skup nizova, niz koji sadrži više nizova.
16 Npr. prazan niz je garbage-collected.
17 Za detaljan opis pogledati Oracle Java Documentation: https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Arrays.html

```
int primjer [] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
```

U ovom slučaju nije potrebno navoditi veličinu niza, jer je veličina očigledna iz samog seta unesenih podataka.
Veličina niza je fiksna u Javi, što znači da se jednom kreiran niz ne može povećavati ili smanjivati. Naravno, neodgovarajuća iteracija niza (npr. ako iterator prelazi veličinu niza) predstavlja grešku u izvršavanju takvog programa:

```
primjer [11]=22; //ispisuje gresku

for (int i=0; i<15; i++) {
neki kôd
} // iteracija preko velicine niza takodjer ispisuje gresku
```

Ako bismo htjeli promjeniti određeni element niza, to se jednostavno postiže pristupanjem njegovom indeksnom elementu i dodjeli nove vrijednosti[18]:

```
primjer [0]=1;
```

Iteracija niza se može postići i "for-each" petljom, koja predstavlja skraćeni oblik "for" petlje i pogodna je za korištenje radi svoje kompaktnosti:

```
String[] mobiteli = {"Samsung", "Apple", "Xiaomi",
"Motorola"};
for (String i : mobiteli) {
  System.out.println(i);
}
```

Vizuelno, niz se može predstaviti na sljedeći način:

Indeksne vrijednosti									
0	1	2	3	4	5	6	7	...	n
Niz[0]	Niz[1]	Niz[2]	Niz[3]	Niz[4]	Niz[5]	Niz[6]	Niz[7]	...	Niz[n]

Tabela 5: Vizuelizacija niza kao strukture podataka

5.1 MINIMUM I MAKSIMUM NIZA

Osnovna prednost nizova je da omogućavaju lako i brzo manipulisanje sa većim setom podataka. Na primjer, idući program nalazi minimum i maksimum elemenata u nizu interacijom:

```
// minimum i maksimum vrijednosti u nizu
public class Main {
```

18 Naravno, ako bi kreirali i manipulisali nizom stringova, koristili bi duple navodnike za svaku takvu manipulaciju. Npr. primjer[0]="Test";

```
    public static void main(String[] args) {
        int nizBrojeva[]=new int[10];
        int min, max;
        nizBrojeva[0]=23;
        nizBrojeva[1]=0;
        nizBrojeva[2]=-546;
        nizBrojeva[3]=2345;
        nizBrojeva[4]=9;
        nizBrojeva[5]=469737;
        nizBrojeva[6]=1342;
        nizBrojeva[7]=-818;
        nizBrojeva[8]=274;
        nizBrojeva[9]=3;

          min = nizBrojeva[0]; /* setuje minimum na prvu
indeksnu vrijednost niza */
            max = nizBrojeva[0]; /* setuje maksimum na prvu
indeksnu vrijednost niza */
           /*min=max=nizBrojeva[0]; Prethodna dva stava mozemo
predstaviti ovom jednolinijskom komandom */
        for(int i=1;i<nizBrojeva.length;i++) {
                    if (nizBrojeva[i]<min) min =nizBrojeva[i];
/*iterise svaku vrijednost i poredi sa varijablom min */
                    if (nizBrojeva[i]>max) max =nizBrojeva[i];
/*iterise svaku vrijednost i poredi sa varijablom max */
        }
        System.out.println("min i max: "+min+ " "+max);
    }
}
```

Napomena:
Program deklariše niz `nizBrojeva` sa dužinom 10. Nizu se dodaju određene numeričke vrijednosti. Deklarišu se int varijable `min` i `max`. Potom se deklariše algoritam koji vraća, odnosno ispisuje minimum i maksimum vrijednosti elemenata ovog niza. Algoritam počinje pretpostavkom da su i minimum i maksimum prva vrijednost niza, odnosno vrijednost sa indeksnim elementom 0. Potom se koristi "for" petlja koja iteriše niz vrijednost po vrijednost (metod `nizBrojeva.length` vraća dužinu niza), te se koriste dva IF uslova. Linijom

```
if (nizBrojeva[i]<min) min =nizBrojeva[i];
```

se postavlja uslov koji se pseudo-kôdom može objasniti kao, ako je vrijednost indeksa i niza `nizBrojeva` manja od varijable `min` (ranije je ta varijabla inicijalizovana sa vrijednošću `nizBrojeva[0]`), dodijeli toj varijabli vrijednost `nizBrojeva[i]`. Analogno, linijom

```
if (nizBrojeva[i]>max) max =nizBrojeva[i];
```

se postavlja uslov koji se pseudo-kôdom može objasniti kao, ako je vrijednost indeksa i niza `nizBrojeva` veća od varijable `max` (ranije je ta varijabla inicijalizovana sa vrijednošću `nizBrojeva[0])`, dodijeli toj varijabli vrijednost `nizBrojeva[i]`.
U suštini, algoritam pretpostavlja da je i minimum i maksimum prvi element niza, pa ako nije tako, kroz iteraciju i provjeru, postavlja novi minimum i maksimum. Linijom

```
min=max=nizBrojeva[0];
```

se jednolinijski može predstaviti dodjeljivanje vrijednosti ovima dvijema varijablama. Na kraju, program jednostavno ispisuje vrijednosti varijabli `min` i `max`.

Ovaj kompletan kôd

```
nizBrojeva[0]=23;
nizBrojeva[1]=0;
nizBrojeva[2]=-546;
nizBrojeva[3]=2345;
nizBrojeva[4]=9;
nizBrojeva[5]=469737;
nizBrojeva[6]=1342;
nizBrojeva[7]=-818;
nizBrojeva[8]=274;
nizBrojeva[9]=3;
```

se može zamijeniti slijedećim:

```
int nizBrojeva[]={23, 0, -546, 2345, 9, 469737,
                  1342, -818, 274, 3};
```

5.2 SORTIRANJE NIZA

Pod sortiranjem niza se podrazumijeva raspoređivanje elemenata po nekom određenom kriterijumu. Iako je prva asocijacija sortiranje numeričkih podataka, mogu se sortirati i neki ostali tipovi podataka, kao npr. stringovi, karakteri. Neki najvažniji algoritmi za sortiranje će biti obrađeni u ovom radu.

5.2.1 BUBBLE SORTIRANJE

Bubble algoritam za sortiranje je jednostavan način koji sortira dati set podataka shodno izabranom kriterijumu, recimo od najmanjeg ka najvećem. Algoritam sortira elemente u više iteracija poredeći početne elemente sa onim već sortiranim.

Ako imamo zadat idući brojčani niz:

```
int [] niz = {62, 38, 22, 19, 20, 10, 95};
```

Prvi prolaz:
(**62, 38,** 22, 19, 20, 10, 95) => (**38, 62,** 22, 19, 20, 10, 95)
Algoritam počinje od prva dva elementa, poredi ih po veličini, uočava da je drugi manji od prvog i mijenja im mjesta.

(38, **62, 22,** 19, 20, 10, 95) => (38, **22, 62,** 19, 20, 10, 95)
Budući da je treći element manji od drugog, mijenja im mjesta.

(38, 22, **62, 19,** 20, 10, 95) => (38, 22, **19, 62,** 20, 10, 95)
Četvrti element postavlja na mjesto trećeg i obrnuto.

(38, 22, 19, **62, 20,** 10, 95) => (38, 22, 19, **20, 62,** 10, 95)
Peti element postavlja na mjesto četvrtog i obrnuto.

(38, 22, 19, 20, **62, 10,** 95) => (38, 22, 19, 20, **10, 62,** 95)
Šesti element postavlja na mjesto petog i obrnuto.

(38, 22, 19, 20, 10, **62, 95**) => (38, 22, 19, 20, 10, **62, 95**)
Nema promjena, budući da je šesti element već manji od sedmog.

Drugi prolaz:
(**38, 22,** 19, 20, 10, 62, 95) => (**22, 38,** 19, 20, 10, 62, 95)
(22, **38, 19,** 20, 10, 62, 95) => (22, **19, 38,** 20, 10, 62, 95)
(22, 19, **38, 20,** 10, 62, 95) => (22, 19, **20, 38,** 10, 62, 95)
(22, 19, 20, **38, 10,** 62, 95) => (22, 19, 20, **10, 38,** 62, 95)
(22, 19, 20, 10, **38, 62,** 95) => (22, 19, 20, 10, **38, 62,** 95)

(22, 19, 20, 10, 38, **62, 95**) => (22, 19, 20, 10, 38, **62, 95**)

Treći prolaz:
(**22, 19,** 20, 10, 38, 62, 95) => (**19, 22,** 20, 10, 38, 62, 95)
(19, **22, 20,** 10, 38, 62, 95) => (19, **20, 22,** 10, 38, 62, 95)
(19, 20, **22, 10,** 38, 62, 95) => (19, 20, **10, 22,** 38, 62, 95)
(19, 20, 10, **22, 38,** 62, 95) => (19, 20, 10, **22, 38,** 62, 95)
(19, 20, 10, 22, **38, 62,** 95) => (19, 20, 10, 22, **38, 62,** 95)
(19, 20, 10, 22, 38, **62, 95**) => (19, 20, 10, 22, 38, **62, 95**)

Četvrti prolaz:
(**19, 20,** 10, 22, 38, 62, 95) => (**19, 20,** 10, 22, 38, 62, 95)
(19, **20, 10,** 22, 38, 62, 95) => (19, **10, 20,** 22, 38, 62, 95)
(19, 10, **20, 22,** 38, 62, 95) => (19, 10, **20, 22,** 38, 62, 95)
(19, 10, 20, **22, 38,** 62, 95) => (19, 10, 20, **22, 38,** 62, 95)
(19, 10, 20, 22, **38, 62,** 95) => (19, 10, 20, 22, **38, 62,** 95)
(19, 10, 20, 22, 38, **62, 95**) => (19, 10, 20, 22, 38, **62, 95**)

Peti prolaz:
(**19, 10,** 20, 22, 38, 62, 95) => (**10, 19,** 20, 22, 38, 62, 95)
(10, **19, 20,** 22, 38, 62, 95) => (10, **19, 20,** 22, 38, 62, 95)
(10, 19, **20, 22,** 38, 62, 95) => (10, 19, **20, 22,** 38, 62, 95)

```
(10, 19, 20, 22, 38, 62, 95) => (10, 19, 20, 22, 38, 62, 95)
(10, 19, 20, 22, 38, 62, 95) => (10, 19, 20, 22, 38, 62, 95)
(10, 19, 20, 22, 38, 62, 95) => (10, 19, 20, 22, 38, 62, 95)
```

Algoritam glasi:

```
 int n = niz.length; /*varijabli n dodjeljujemo dužinu niza
                       radi lakšeg manipulisanja */

         for (int i = 0; i < n-1; i++) // prvi prolaz čitavim
nizom
            for (int j = 0; j < n-i-1; j++) /*pod-prolaz
                                    */nesortiranim ostatkom
niza
                if (niz[j] > niz[j+1])
                {
                    // zamijeniti niz[j+1] and niz[j]
                    int temp = niz[j];
                    niz[j] = niz[j+1];
                    niz[j+1] = temp;
                }
```

Napomena:
Predstavljena je jedna od mogućih varijanti bubble sort algoritma. Na početku deklarišemo varijablu n, kojoj dodjeljujemo vrijednost dužine niza (nije neophodno ali time izbjegavamo pisanje `niz.length` u daljem kôdu). Koristeći "for" petlju, iterišemo niz od nultog do posljednjeg indeksa, povećavajući brojač na kraju svake iteracije. Podpetljom iterišemo prelaze koji su preostali, odnosno dio nesortiranih elemenata (otud izraz `j < n-i`[19]`-1`). Ostatak kôda je dalje jasan, ukoliko je iterisani element veći od istog tog elementa uvećanog za jedan (`if (niz[j] > niz[j+1])`, vršimo njegovu zamjenu koristeći pomoćnu varijablu temp.

Analiza vremenske složenosti

Bubble sort je kvadratni algoritam, budući da se sastoji iz dvije petlje, od kojih se prva izvršava i puta (odnosno n-1 puta), a druga, pod-petlja, n-2 puta[20]. Ukupno vrijeme izvršavanja ovog algoritma se može izraziti kao:

```
for (int i = 0; i < n-1; i++) // izvrsava se n-1 puta
     for (int j = 0; j < n-i-1; j++) // izvrsava se n-2 puta
          if (niz[j] > niz[j+1]) /* zadnja dva reda uzimaju
               zamjena elemenata */ vrijednost neke konstante
               c

T(n) = (n-1) (n-2) c
```

19 Varijabla i označava broj već sortiranih elemenata i zato se oduzima, kako bi se u podpetlji sortirali samo preostali elementi.

20 Podpetlja se izvršava za jedan broj manje od glavne petlje, jer u krajnjoj iteraciji, u primjeru kada je i=6, uslov `j < n-i-1` uslov postaje nula, odnosno ta iteracija ne postoji.

$T(n) = cn^2 + 3cn + 2c$

Odnosno, ovaj algoritam u O-notaciji ima vremensku kompleksnost (vrijeme izvođenja) n^2.

5.2.2 INSERT SORTIRANJE

Algoritam pomoću kojeg se dati niz može sortirati slično kao što igrač bridža sortira karte naziva se insert sortiranje. Kao i kod bubble sortiranja, niz se logički dijeli na sortirani i nesortirani dio, te se iterativno korak po korak sortiraju elementi shodno određenom kriterijumu (takođe uzimamo primjer od najmanjeg ka najvećem elementu).

Vratimo se nizu koga smo ranije deklarisali:

```
int [] niz = {62, 38, 22, 19, 20, 10, 95};
```

Prvi prolaz:
(**62, 38,** 22, 19, 20, 10, 95) => (**38, 62,** 22, 19, 20, 10, 95)
Algoritam počinje sa datim nizom, poredeći prvi element koji ima prethodnika sa njim. Ako je manji mijenja im mjesta.

Drugi prolaz:
(**38, 62, 22,** 19, 20, 10, 95) => (**22, 38, 62,** 19, 20, 10, 95)
Ako je prethodni element manji, algoritam poredi pretposljednji element (dakle sve prethodne), i mijenja im mjesta.

Treći prolaz:
(**22, 38, 62, 19,** 20, 10, 95) => (**19, 22, 38,** 62, 20, 10, 95)

Četvrti prolaz:
(**19, 22, 38, 62, 20,** 10, 95) => (**19, 20, 22, 38, 62,** 10, 95)

Peti prolaz:
(**19, 20, 22, 38, 62, 10,** 95) => (**10, 19, 20, 22, 38, 62,** 95)

Šesti prolaz:
(**10, 19, 20, 22, 38, 62, 95**) => (**10, 19, 20, 22, 38, 62, 95**)

Algoritam glasi:

```
int n = niz.length;
        for (int i = 1; i < n; ++i) {
            int indeks = niz[i];
            int j = i - 1;

            /* Pomjera elemente niz[0..i-1], vecih od indeks,
               jednu poziciju naprijed od
```

```
            njihove trenute pozicije */

        while (j >= 0 && niz[j] > indeks) {
            niz[j + 1] = niz[j];
            j = j - 1;
        }
        niz[j + 1] = indeks;
    }
```

Napomena:
Algoritam koristi "for" petlju u okviru koje deklariše varijablu indeks i dodjeljuje joj, po iteraciji, vrijednost tekućeg indeksa niza, počev od indeksne vrijednosti 1 do n (dužina niza). Varijabla j predstavlja vrijednost iteratora umanjenu za jedan. Ako bismo unutar petlje ispisali vrijednost varijable indeks, dobili bi vrijednosti: 38, 22, 19, 20, 10, 95, što ne znači da je u pitanju varijabla sa više vrijednosti što je nemoguće, već da u svakoj iteraciji se toj varijabli dodjeljuje indeksna vrijednost i iterisanog niza niz. Pod-petljom "while" se postavljaju dva logička uslova (`j >= 0 && niz[j] > indeks`), i uz uslov da su oba tačna, element sa većom vrijednošću se stavlja naprijed, te brojač petlje se umanjuje za jedan. Petlja "while" potom završava i linijom `niz[j + 1] = indeks;` vraćamo vrijednosti elemenata iterisanih i snimljenih u varijabli indeks na indeksno mjesto niza `[j + 1]`.

Analiza vremenske složenosti

U najgorem slučaju, insert sortiranje je kvadratni algoritam, budući da imamo dvije petlje, od kojih se "for" petlja izvršava n puta, te "while" petlja n-1 puta.

$T(n) = n\ (n-1) + c$
$T(n) = n^2 - n + c$

Ukoliko imamo već sortirani niz, "while" petlja se izvršava 0 puta, odnosno ne izvršava se, pa se kompleksnost svodi na n.

Odnosno, ovaj algoritam u O-notaciji ima vremensku kompleksnost (vrijeme izvođenja) u najgorem slučaju n^2 a u najboljem n.

5.2.3 SELECTION SORTIRANJE

Treći algoritam koji ćemo obraditi u ovom radu sortira niz tako što konstantno nalazi minimalni element iz nesortiranog dijela niza i stavlja ga na početak niza. Dakle i ovaj algoritam, kao i prethodna dva, u nizu sadržava sortirani i nesortirani dio. U svakoj iteraciji, minimum iz nesortiranog niza se stavlja u sortirani dio niza.

Kao i do sad, koristićemo niz:

```
int [] niz = {62, 38, 22, 19, 20, 10, 95};
```

Prvi prolaz:
(62, 38, 22, 19, 20, 10, 95) => (**10**, 38, 22, 19, 20, 62, 95)

Drugi prolaz:
(10, 38, 22, 19, 20, 62, 95) => (**10, 19**, 22, 38, 20, 62, 95)

Treći prolaz:
(10, 19, 22, 38, 20, 62, 95) => (**10, 19, 20**, 38, 22, 62, 95)

Četvrti prolaz:
(10, 19, 20, 38, 22, 62, 95) => (**10, 19, 20, 22**, 38, 62, 95)

Peti prolaz:
(10, 19, 20, 22, 38, 62, 95) => (**10, 19, 20, 22, 38, 62, 95**)
U ovoj iteraciji nema promjena budući da su preostali elementi već sortirani.

Algoritam:

```
int n = niz.length;

        for (int i = 0; i < n-1; i++)
        {
            // Nalazi minimum u nesortiranom dijelu niza
            int min = i;
            for (int j = i+1; j < n; j++)
                if (niz[j] < niz[min])
                    min = j;

            // Mijenja minimalni element sa prvim elementom

            int temp = niz[min];
            niz[min] = niz[i];
            niz[i] = temp;
        }
```

Napomena:
Algoritam "for" petljom iteriše sve elemente niza, te za dati element slično kao bubble sort podpetljom iteriše nesortirani dio niza. Ukoliko dati element ispunjava logički uslov `if (niz[j] < niz[min])`, varijabli `min` se dodjeljuje vrijednost interatora `j`. Potom se vrši se zamjena pozicija elemenata koristeći varijablu `temp`.

Analiza vremenske složenosti

Kao i prethodni algoritmi, budući da se radi o višestrukim petljama, selection algoritam za sortiranje je kvadratni algoritam, odnosno u O-notaciji ima vremensku kompleksnost (vrijeme izvođenja) n^2.

5.3 PRETRAGA NIZA

5.3.1 LINEARNA PRETRAGA

Jedna od osnovnih operacija u radu sa nizovima je pretraga njihovih elemenata. Pretraga omogućava rješavanje čitavog niza problema, jer nam dodaje novu funkcionalnost, pogotovo ako koristimo nizove kao mjesto skladištenja nekog seta podataka. Ako želimo za dati niz pretražiti njegove elemente da bi našli neku vrijednost, kao i njenu poziciju u nizu, najjednostavniji metod za to je linearna pretraga. Linearnom pretragom iterišemo niz od nultog do posljednjeg elementa, te svaki element poredimo sa traženim. Ako je uslov tačan, vraćamo taj element. Prikažimo primjer ovog algoritma koristeći niz brojeva koji smo već ranije koristili.

Primjer algoritma[21]:

```
//linearna pretraga niza
public class Main {

    public static void main(String[] args) {
        int[] niz = {62, 38, 22, 19, 20, 10, 95};
           int x=20; //deklaracija varijable ciju vrijednost
trazimo

     int rezultat = pretraga(niz, 20); /*varijabli rezultat
     pridruzujemo metod pretraga koga pozivamo sa dva
parametra */

     if (rezultat==0)
          System.out.println("Trazeni    broj    "+x+"    nije
pronadjen");
     else {
          System.out.println("Trazeni broj je pronadjen kao
indeks         sa vrijednosti "+x);
          }
     }

     public static int pretraga(int [] niz, int x) {
```

21 Ovdje je naveden kompletan program, ne samo algoritam, jer prikazani algoritam zapravo metod koji se poziva u glavnom main metodu.

```
        int n = niz.length;
        for(int i=0; i<n; i++)
            if(niz[i]==x)
                return x;
return 0;

          }
     }
```

Napomena:
Algoritam je definisan u metodu pretraga, kojeg pozivamo sa dva parametra, niz i varijablom `x` koja predstavlja broj koji pretražujemo. Niz se iteriše i za svaku iteraciju se provjerava da li je jednaka vrijednosti varijable `x`. Ako je jednaka, metod vraća tu vrijednost, ako nije, vraća 0. U main metodu pozivamo metod `pretraga`, dodjeljujemo njegovu vrijednost varijabli `rezultat`, te jednostavnim logičkim uslovom ispisujemo dobijeni rezultat. Ovaj algoritam u O-notaciji ima vremensku kompleksnost (vrijeme izvođenja) n, budući da se sastoji iz jedne petlje i konstantne vrijednosti.

5.3.2 BINARNA PRETRAGA

Ako želimo da pretražimo sortiran niz, možemo iskoristiti algoritam koji se zove binarna pretraga. Ovaj algoritam u iteracijama dijeli traženi niz na polovinu. Na početku se bira srednji element niza te procjenjuje da li je taj element veći ili manji od tražene vrijednosti. Ako je manji, onda se bira gornja polovina i iteriše u narednom prolazu, te tako redom sve dok se tražena vrijednost nađe, odnosno ne nađe.

Za primjer ćemo koristiti već iz ranijih primjera poznat niz:

```
int[] niz = {62, 38, 22, 19, 20, 10, 95};
```

```
koji je za potrebe ovog algoritma sortiran:
```

```
(10, 19, 20, 22, 38, 62, 95)
```

Pretpostavimo da tražimo vrijednost 19.

Prvi prolaz:

(10, 19, 20, **22**, 38, 62, 95) => (10, 19, 20)
Srednja vrijednost veća od tražene, odabira se prva polovina niza.

Drugi prolaz:

(10, **19**, 20) => **19**

Traženi element pronađen.

Primjer algoritma:

```
//binarna pretraga niza
public class Main {

    public static void main(String[] args) {

        int niz[] = {10, 19, 20, 22, 38, 62, 95};
        int n = niz.length;
        int x = 19;
        int rezultat = binarnaPretraga(niz, x);
        if (rezultat == 0)
            System.out.println("Trazeni element ne postoji");
        else
            System.out.println("Element nadjen na poziciji "
                    + "index " + rezultat);
    }

    public static int binarnaPretraga(int niz[], int x)
    {
        int l = 0, r = niz.length - 1;
        while (l <= r) {
            int m = l + (r - l) / 2;

            //Provjerava da li je x prisutan na sredini
            if (niz[m] == x)
                return m;

            //Ako je x veci, ignorise lijevu polovinu
            if (niz[m] < x)
                l = m + 1;

                //Ako je x manji, ignorise desnu polovinu
            else if (niz[m] > x)
                r = m- 1;

        }

        /* ako program dodje do ovdje
         onda trazenog elementa nema */
        return 0;
    }
}
```

Napomena:
Algoritam binarne pretrage je sadržan u metodu `binarnaPretraga`. Deklarisane su dvije varijable, `l` i `r`, koje olakšavaju dalje pisanje logike. "While" petljom[22] se provjerava logički

22 Mogli smo koristiti i "for" petlju ali ovdje je "while" pogodnija jer uzima manje argumenata.

uslov da li je varijabla `l`, odnosno brojač, manja od dužine niza umanjene za jedan (što predstavlja opseg iterisanja petlje). Ako jeste, definiše se varijabla `m` takva da se nalazi na polovini niza. Ako je taj element m jednak varijabli `x` (za čiju vrijednost provjeravamo niz), metod vraća taj element. Ako je vrijednost `x` veća, program ignoriše lijevu polovinu i uvećava brojač za jedan, te konačno, ako je vrijednost `x` manja, program provjerava lijevu polovinu niza i umanjuje brojač.

Analiza vremenske složenosti

Budući da binarna pretraga koristi već sortiran niz, u prvoj iteraciji dužina pretrage iznosi n, u drugoj n/2, trećoj n/4 i tako dalje, odnosno uopšteno se može reći, za m-tu iteraciju, dužina pretrage iznosi $n/2^{m-1}$. Za posljednju iteraciju važi $n/(2^{m-1})=1$, budući da preostaje samo jedan element, odnosno u pitanju je logaritamski algoritam koji u O-notaciji ima vremensku kompleksnost (vrijeme izvođenja) log n.

ZAKLJUČAK

Sa pojavom informatičkog društva i treće industrijske revolucije (već se uveliko govori i o četvrtoj), rapidno se mijenja svijet oko nas, a promjene koje donosi tehnologija su takve da ih lako uočavamo i u svakodnevnom životu. Mijenjaju se načini razmišljanja, obavljanja poslova, svakodnevni život, praktično, ova industrijska revolucija remeti skoro svaku djelatnost od značaja ne samo u datom društvu već i na globalnom nivou. Algoritmi svakim danom dobijaju na značaju obzirom da se primjenjuju u oblastima kao što su:

- vještačka inteligencija,
- nanotehnologija,
- biotehnologija,
- kvantno računarstvo,
- nastajanje novih materijala,
- autonomna vozila,
- robotika,
- 3D štampanje.

Vještačka inteligencija je već svuda oko nas: rapidno se razvijaju samoupravljajuća vozila, virtuelni asistenti, programi za simultano prevođenje. U samom središtu svih ovih tehnologija se nalaze algoritmi kao osnovne jedinice svih ovih računarskih operacija. Prema tome, sigurno je da je izučavanje algoritama jedna od najznačajnijih investicija koju pojedinac u 21. vijeku može da načini u smislu ličnog usavršavanja.

U ovom radu su obrađeni najosnovniji načini primjene nekih jednostavnijih algoritama u programskom jeziku Java. Dotakli smo se vremenske kompleksnosti algoritama, u kome je bilo riječi o vremenskom trošku izvođenja algoritama. Ukratko je obrađena najosnovnija sintaksa potrebna za implementaciju jednostavnijeg programskog kôda, pružen osnovni uvid u neke od struktura podataka u Javi kao što su tipovi podataka, nizovi, te nekih drugih programskih segmenata neophodnih za izvršavanje programa kao što su petlje. Algoritmi koji su predstavljeni su sortiranje niza, odnosno bubble, insert i selection sortiranje, te pretraga niza, linearna i binarna pretraga. Za sve algoritme su dati pseudo-kôdovi kao i kompletan programski kôd u programskom jeziku Java.

LITERATURA

[1] Core Java, Volume I-Fundamentals, Eleventh Edition, Cay. S. Horstmann, Pearson, 2019

[2] Data Structures&Algorithms in Java, Sixth Edition, Michael Goodrich, Roberto Tamassia, John

Wiley&Sons, Inc., 2014

[3] Introduction to Algorithms, Third Edition, Thomas Cormen, Charles Leiserson, Ronald Rivest,

Clifford Stein, MIT Press, 2009

[4] Java A Begginer's Guide, Eight Edition, Herbert Schildt, McGraw-Hill, 2019

[5] Java All-In-One, 11th Edition, Doug Lowe, John Wiley&Sons, 2020

[6] Lecture Notes for Data Structures and Algorithms, John Bullinaria, University of Birmingham, 2019

[7] Thinking in Java, Fourth Edition, Bruce Eckel, Prentice Hall 2006

[8] https://docs.oracle.com/javase/specs/jls/se15/html/index.html, The Java Language Specification, Java SE 15, pristupano 25.04.2021

[9] Uvod u algoritme i strukture podataka, Dejan Živković, Univerzitet Singidunum, 2018