Algoritmi sortiranja

Autori:
Milan Segedinac
Goran Savić

1. Algoritam i algoritamska složenost

Algoritam je opis načina rešavanja problema koji se sastoji od konačno mnogo koraka. *Specifikacijom problema* određuje se željena veza između *ulaza* i *izlaza*, a algoritmom se navode *konkretne procedure* (koraci) kojima se ta veza ostvaruje kao transformacija ulaza u izlaz.

Algoritmi se mogu zadati u različitim notacijama, kao što su:

- *Prirodni jezik*
- *Pseudokod* – neformalni opis visokog nivoa koji podseća na implementaciju u nekom (često hipotetskom) programskom jeziku
- *Blok dijagram* – dijagram pomoću kog se grafički predstavlja tok izvršavanja algoritma
- *Programski kod*

Primer – Euklidov algoritam

Euklidov algoritam je efikasan metod za pronalaženje *najvećeg zajedničkog delioca* (NZD) dva cela broja. Obzirom na njegovu jednostavnost i istorijsku važnost, mi ćemo ovaj algoritam iskoristit za ilustrovanje načina na koji se algoritmi mogu zadati.

Specifikacija problema

Neka su data dva cela broja, a i b, takva da važi $a \geq b \geq 0$. Pronaći najveći prirodni broj c koji brojeve a i b deli bez ostatka. Ulazi ovog algoritma su brojevi a i b, a izlaz je NZD c.

Opis Euklidovog algoritma – algoritam zadat prirodnim jezikom

Algoritam se odvija sukcesivnim ponavljanjem sledećih koraka, sve dok b ne bude imalo vrednost 0:
Ako je $b \geq a$, onda b dobija vrednost $b-a$.
Ako je $a > b$, onda a dobija vrednost $a-b$.
Kada je $b = 0$, onda je a NZD za brojeve sa ulaza

Pseudokod Euklidovog algoritma

Euklidov algoritam zadat je pseudokodom na listingu ispod.

```
initialize: A := a, B := b
  while B ≠ 0 do
    if B ≥ A then
      B := B – A
    else
      A := A - B
    end(if)
  end(while)
return A
```

Listing – Pseudokod Euklidovog algoritma

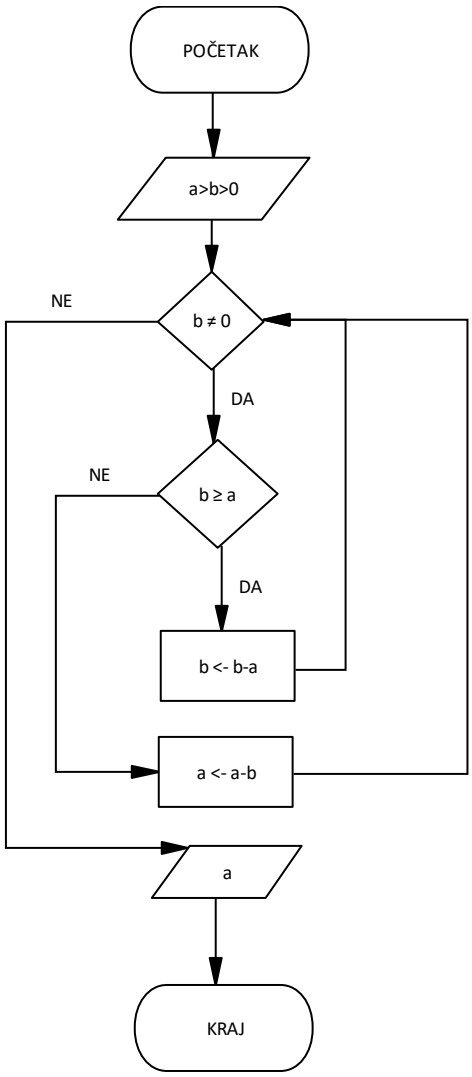
Glavna pogodnost zadavanja algoritama pomoću pseudokoda u odnosu na govorni jezik je u tome što se algoritam zadaje preciznije, bez višeznačnosti koje mogu biti posledica prirodnog jezika. Čak i ako čitalac ne zna detalje sintakse pseudokoda, zahvaljujući tome što zna da programira u bilo kom programskom jeziku razumeće ovako zadate algoritme.

Algoritmi zadati pseoudokodom se ne mogu izvršavati, što je njihov osnovni nedostatak u odnosu na algoritme zadate njihovom implementacijom u nekom programskom jeziku. Međutim, programski

kod često sadrži detalje implementacije koji skreću pažnju sa samog algoritma, tako da algoritam zadat pseudokodom može da budu razumljiviji od programskog koda kojim je algoritam implementiran.

Blok dijagram Euklidovog algoritma

Blok dijagram Euklidovog algoritma dat je slikom ispod.



Slika – Blok dijagram Euklidovog algoritma

Blok dijagrami mogu da budu pogodni za prikazivanje jednostavnih algoritama. Međutim, kako algoritmi postaju složeniji, blok dijagrami su sve manje pregledni i teži za razumevanje u odnosu na, recimo, pseudokod.

Implementacija Euklidovog algoritma

Listingom ispod data je implementacija Euklidovog algoritma u programskom jeziku Java

```
package vp.algoritmisortiranja;

public class EuklidovAlgoritam {

	public static void main(String[] args) {
		int a = 25;
		int b = 40;
		while(b!=0){
			if(b>=a){
				b -= a;
			}
			else{
				a -= b;
			}
		}
		System.out.println(a);
	}
}
```

Listing – Implementacija Euklidovog algoritma u programskom jeziku Java

Prednosti zadavanja algoritma u ovom formatu su u tome što je format vrlo detaljan i što program možemo i da izvršimo. Nedostatak je u tome što zahteva da čitalac zna detalje programskog jezika u kom je algoritam implementiran. U ovom primeru potrebno je da zna šta znači `System.out.println` ili `public static void main(String[] args)` što su tehnički detalji programskog jezika Java.

Složenost algoritama

Za jednu specifikaciju problema često možemo definisati različite algoritme koji obezbeđuju željenu transformaciju ulaza u izlaz. Posmatraćemo problem sortiranja niza celih brojeva. Problem možemo specificirati na sledeći način: nesortiran niz celih brojeva (`nNiz`) dat je kao ulazni podatak, a na izlazu treba da se dobije niz koji ima iste vrednosti kao i `nNiz`, samo poređane od najmanje ka najvećoj (`sNiz`).

Selection sort

Niz bismo mogli da sortiramo tako što ćemo proći od prvog elementa do poslednjeg, pronaći najmanji element u nizu i prebaciti ga na prvo mesto, a prvi element na njegovo. Zatim ćemo proći kroz ostatak niza (od drugog do poslednjeg elementa), pronaći najmanji element i zameniti ga sa drugim elementom. To ćemo ponavljati sve dok ne stignemo do kraja niza. Ovaj algoritam sortiranja zove se *selection sort* i opis ovog algoritma pseudokodom dat je listingom ispod.

```
Initialize: sNiz <- nNiz
for i in 0 -> sNiz.length - 2 do
  minIndex = i
  for j in (i + 1) -> (sNiz.length - 1) do
    if sNiz[j] < sNiz[minIndex] then
      minIndex = j
    end(if)
    swap(sNiz[i], sNiz[minIndex])
  end(for)
end(for)
```

Listing – Selection sort algoritam

Bucket sort algoritam

Pretpostavimo da će niz od *n* elemenata koji sortiramo nakon sortiranja početi vrednošću *0*, a završiti sa *n-1* i da će ga činiti sekvencijalni celi brojevi (da u nizu neće biti ponovljenih vrednosti ni „rupa“), odnosno da ulazni niz sadrži sve cele brojeve između 0 i n-1. Takav niz bismo mogli da sortiramo i tako što ćemo napraviti prazan niz (`sNiz`) iste dužine kao i nesortirani niz (`nNiz`). Zatim ćemo uzeti prvi element `nNiz`-a i staviti ga na poziciju `i` u `sNiz`-u gde je `i=nNiz[0]`. Zatim ćemo uzeti sledeći element iz `nNiz`-a i postaviti ga na poziciju `i` u `sNiz`-u gde je `i=nNiz[1]`. Ovo ćemo ponavljati dok ne stignemo do kraja `nNiz`-a. Ovaj algoritam sortiranja se zove *bucket sort* i opisan je pseudokodom u listingu ispod.

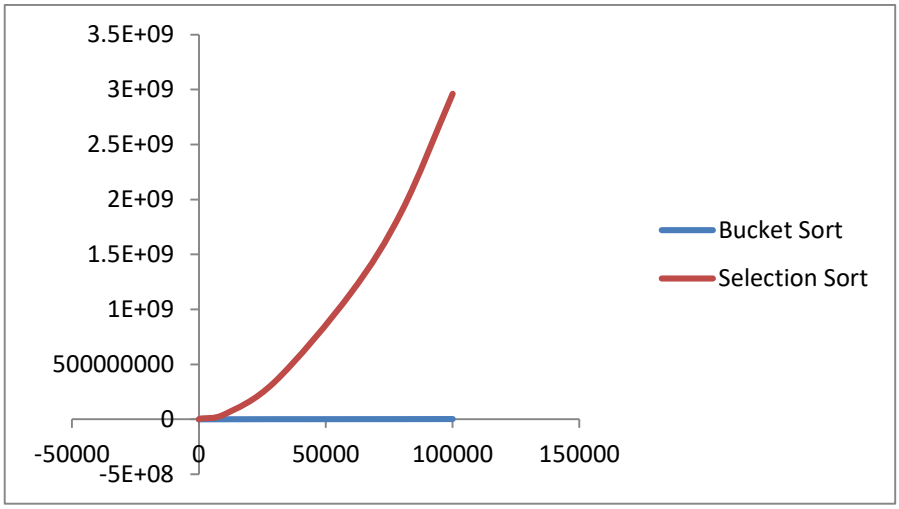
```
Initialize: sNiz = int[nNiz.length]
for i in 0 -> sNiz.length - 1 do
  sNiz[nNiz[i]]=nNiz[i]
end(for)
```

Listing – Bucket sort algoritam

Ako ih primenimo u situacijama u kojim je bucket sort algoritam adekvatan, rezultat izvršavanja programa koji implementiraju ova dva algoritma će biti istovetan – dobiće se sortiran niz. Ali između izvršavanja ova dva programa će postojati značajna razlika. Da bismo tu razliku uočili, implementirali smo ova dva algoritma u programskom jezuku Java i simulirali njihovo izvršavanje na nizovima različite dužine. Pri tome su dobijeni rezultati prikazani tabelom i grafikom ispod. Prva kolona tabele (n) odnosi se na broj elemenata ulaznog niza. Rezulati iz tabele prikazani su grafikom ispod. Na *x* osi je broj elemenata ulaznog niza, a na *y* osi je vreme izršavanja u sekundama.

N	Bucket Sort (ms)	Selection Sort (ms)
10	0.46	0.38
100	0.47	0.53
1000	0.84	7.74
10000	1.17	44
30000	1.95	342.84
60000	2.19	1151.43
80000	2.55	1896.10
100000	3.17	2960.83

Tabela – vreme izvršavanja *bucket sort* i *selection sort*



Dijagram – vreme izvršavanja *bucket sort* i *selection sort*

Iz izmerenih podataka može se uočiti da se vreme potrebno za izvršavanje *selection sort* algoritma značajno brže povećava nego vreme izvršavanja bucket sort algoritma kako raste broj elemenata niza. To nije posledica implementacije nego prirode samih algoritama i nije potrebno da implementiramo program da bismo mogli da procenimo koliko je algoritam složen – to možemo da znamo na osnovu *analize* algoritma. Razumevanje složenosti algoritma veoma je korisno kada treba da se opredelimo koji algoritam ćemo implementirati za rešavanje nekog problema, ili čak da li ćemo uopšte i rešavati taj problem.

Analiza algoritama

Analiza algoritma omogućuje da saznamo koliko *resursa* će izvršavanje posmatranog algoritma zahtevati. Najčešće će nas interesovati koliko *procesorkog vremena* će biti potrebno za izvršavanje algoritma, ali u pojedinim situacijama i drugi resursi, kao što su *zauzeće memorije* ili *mrežni protok* mogu biti predmet analize algoritama.

Analiza algoritama se ne bavi efikasnošću algoritma u nekoj konrektnoj tehnologiji, već efikasnošću algoritama *uopšte*. Stoga sve algoritme posmatramo kao da se izvršavaju na *modelu* računara koji se zove *mašina slučajnog pristupa* (eng. random-access machine, RAM[1]). U RAM modelu podržane su tipične aritmetičke operacije (sabiranje, oduzimanje, množenje, deljenje), dodele vrednosti varijablama i kontrolne operacije (uslovno izvršavanje, pozivi procedura) i za svaku od ovih *primitivnih operacija* se smatra da zahteva jednaku količinu procesorskog vremena. Ovo je, naravno, gruba aproksimacija, ali nam omogućava da složenost algoritma jednostavno izrazimo *brojem primitivnih operacija*. Pored toga, praksa pokazuje da je složenost algoritma na RAM modelu odličan prediktor performanse implementacije tog algoritma u konkretnim tehnologijama.

Kao što smo videli na simulacijama izvršavanja, vreme potrebno za izvršavanje algoritma sortiranja zavisi od broja elemenata u nizu koji se sortira. Složenost selection sort algoritma možemo odrediti tako što ćemo *prebrojati* koliko puta će se svaka linija pseudokoda algoritma izvršiti i proceniti broj primitivnih operacija potrebnih za izvršavanje svake linije pseudokoda.

[1] Random-access machine model računara ne treba brkati sa Random-access memorijom. Slučajno imaju iste akronime.

Operacija	Cena operacije	Broj ponavljanja
`sNiz = nNiz`	c_1	*1*
`for i in 0 -> sNiz.length - 2 do`	c_2	*n-1*
`  minIndex = i`	c_3	*n-1*
`  for j in (i + 1) -> (sNiz.length - 1) do`	c_4	*n-1*
`    if sNiz[j] < sNiz[minIndex]`	c_5	*(n-1)*(n-1)/2*
`    minIndex = j`	c_6	*(n-1)*(n-1)/2*
`  swap(sNiz[i], sNiz[minIndex])`	c_7	*n-1*

Listing – Analiza selection sort algoritma

Ako $c_1, \ldots, c_7$ predstavljaju brojeve primitivnih operacija koje je potrebno izvršiti za svaku liniju pseudokoda, onda je vreme potrebno za izvršavanje selection sort algoritma:

$$T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5\frac{(n-1)^2}{2} + c_6\frac{(n-1)^2}{2} + c_7(n-1)$$

$$= \left(c_1 - c_2 - c_3 - c_4 + \frac{c_5}{2} + \frac{c_6}{2} - c_7\right) + (c_1 + c_2 + c_3 + c_4 - c_5 - c_6 + c_7)n + (\frac{c_5}{2} + \frac{c_6}{2})n^2$$

Ovakva procena složenosti algoritma je komplikovana, a često nepotrebno precizna. Najčešće možemo da zanemarimo sve manje bitne članove ove funkcije i da posmatramo samo vodeći (u našem slučaju, to je $(\frac{c_5}{2} + \frac{c_6}{2})n^2$. Takođe, za ovaj član možemo da zanemarimo koeficijent $(\frac{c_5}{2} + \frac{c_6}{2})$ i zaključimo da se selection sort algoritam izvršava u *kvadratnom vremenu*, što zapisujemo kao $\Theta(n^2)$. Do ovog zaključka mogli smo doći i bez detaljne analize algoritma – dovoljno je bilo da vidimo da imamo *ugnježdenu petlju*.

Kada smo za algoritam odredili da je njegova složenost $\Theta(n^2)$ to znači da postoje konstante c_1 i c_2 takve da će vreme izvršavanja algoritma uvek biti veće od c_1n^2 i uvek biti manje od c_2n^2, odnosno pronašli smo funkciju $f(n)=n^2$ takvu da važi $c_1n^2 < T(n) < c_2n^2$.

Često nam treba samo gornja granica složenosti algoritma, odnosno funkcija f takva postoji konstanta c za koju važi $T(n) < cf(n)$, pri čemu zanemarujemo donju granicu složenosti algoritma. Ova mera složenosti algoritma izražava se kao *O(f(n))*, i zove se *veliko O notacija* (eng. *big-O*).

Tabelom ispod dat je pregled funkcija vremena izvršavanja algoritama, od najmanje složenih ka najsloženijim.

T(n)	Vreme izvršavanja	Kategorija algoritama
$\Theta(c)$	Konstantno	Efikasni
$\Theta(logn)$	Logaritmsko	
$\Theta(n)$	Linearno	
$\Theta(nlogn)$	Linearno logaritamsko	
$\Theta(n^c)$	Polinomijalno (npr. kvadratno)	
$\Theta(c^n)$	Eksponencijalno	Neefikasni
$\Theta(n!)$	Faktorijelsko	

Tabela – vreme izvšavanja algoritama

Algoritme smatramo efikasnim ukoliko se izvršavaju (u najgorem slučaju) u polinomijalnom vremenu. Algoritme koji se izvršavaju u eksponencijalnom ili, još gore, u faktorijelskom vremenu

smatramo neefikasnim i praktično neupotrebljivim. Algoritam čija je složenost $\Theta(2^n)$ koji bi se izvršavao na računaru koji obavlja 10^{12} operacija u sekundi bi se, za n=100 (relativno mali broj elemenata), izvršio za $4*10^{10}$ godina, što je reda veličine starosti univerzuma.