Sadržaj

1 O algoritmima i složenosti **7**

1.1 Pojam algoritma 7
1.2 Zapisivanje algoritama 9
1.3 Složenost algoritma 10
1.3.1 Analiza složenosti metode za računanje minimuma niza . . 11
1.3.2 Analiza složenosti sortiranja niza primenom sortiranja selekcijom 13
1.3.3 Analiza složenosti sortiranja niza primenom sortiranja umetanjem 14
1.4 Asimptotska notacija 17
1.4.1 Asimptotska oznaka Θ 17
1.4.2 Asimptotska oznaka O 19
1.4.3 Asimptotska oznaka Ω 20
1.4.4 Asimptotske oznake o i ω 21
1.4.5 Neka svojstva asimptotskih oznaka 22
1.4.6 Master teorema 23
1.5 Asimptotska složenost algoritama 25

2 Liste, Stekovi, Redovi **29**

2.1 Liste 29
2.1.1 Implementacija liste zasnovana na nizu 31
2.1.2 Implementacija liste zasnovana na povezanim listama . . . 33
2.1.3 Poređenje implementacija 40
2.1.4 Dvostruko povezane liste 41
2.2 Stek 44
2.2.1 Implementacija zasnovana na nizu 44
2.2.2 Implementacija zasnovana na povezanim listama 45
2.2.3 Primer primene steka 46
2.3 Red 53
2.3.1 Implementacija reda zasnovana na nizu 53
2.3.2 Implementacija reda zasnovana na povezanim listama . . . 55
2.3.3 Ilustracija primene reda 56
2.4 Red sa dva kraja 59

2.4.1 Implementacija reda sa dva kraja bazirana na nizovima 59
2.4.2 Implementacija reda sa dva kraja bazirana na povezanoj listi 61
2.4.3 Ilustracija primene reda sa dva kraja 62
2.5 Red prioriteta 64

3 Binarna stabla **69**
3.1 Apstraktni tip podataka koji reprezentuje čvor 71
3.2 Obilazak stabla 71
3.3 Implementacija binarnih stabala 75
3.3.1 Implementacija zasnovana na referencama (pokazivačima) 75
3.3.2 Memorijski zahtevi implementacije bazirane na referencama 78
3.4 Implementacija kompletnog binarnog stabla pomoću niza 80
3.5 Binarno pretraživačko stablo 81
3.5.1 Pronalaženje podatka u binarnom pretraživačkom stablu . . 83
3.5.2 Dodavanje podatka u binarno pretraživačko stablo 84
3.5.3 Izbacivanje podatka sa zadatim ključem iz binarnog pretraživačkog stabla 85
3.6 Hip 87
3.6.1 Formiranje hip-a od date kolekcije podataka 89
3.6.2 Izbacivanje najveće vrdenosti iz maks-hip-a 92
3.6.3 Implementacija reda prioriteta pomoću maks-hip-a 92
3.7 Hafmanovo stablo kodiranja, Hafmanovo kodiranje 92
3.7.1 Dokaz korektnosti postupka za obrazovanje optimalnog stabla 99

4 Uopštena (generalizovana) stabla **103**
4.1 Apstraktni tip podataka za uopšteno stablo 104
4.2 Obilazak stabla 105
4.3 Implementacije interfejsa za uopšteno stablo 106
4.3.1 Implemementacija zasnovana na listi dece 107
4.3.2 Implementacija Levo–Dete/Desno-Brat 107
4.3.3 Implementacija bazirana na „dinamički” kreiranim čvorovima 109
4.3.4 Implementacija Levo–Dete/Desno–Brat bazirana na „dinamički” kreiranim čvorovima 111
4.4 Reprezentacija strukture `Union/Find` 112
4.4.1 Primer primene strukture `Union/Find` 117
4.4.2 Još jedan primer primene strukture `Union/Find` 118

5 Balansirana binarna stabla **123**
5.1 AVL stablo . . . 123
5.1.1 Rotacija . . . 126
5.1.2 Dodavanje nove vrednosti u AVL stablo . . . 128
5.1.3 Brisanje vrednosti iz AVL stabla . . . 131

6 B-stabla **135**
6.1 B-stablo . . . 136
6.1.1 Implementacija čvora B-stabla . . . 137
6.1.2 Umetanje novog podatka u B-stablo . . . 139
6.1.3 Brisanje vrednosti iz B-stabla . . . 144
6.1.4 Umetanje u B-stablo sa ažuriranjem odozgo-nadole . . . 152
6.1.5 Brisanje podatka iz B-stabla sa ažuriranjem odozgo-nadole 155
6.2 B*-stablo . . . 158
6.3 B+-stablo . . . 159
6.4 2–3–4 stabla . . . 161
6.5 2–3 stabla . . . 166

7 Sortiranje **167**
7.1 Sortiranje umetanjem . . . 167
7.2 Sortiranje selekcijom . . . 168
7.3 Bubble sortiranje . . . 169
7.4 Quick sort . . . 171
7.4.1 Analiza složenosti Quck sort–a . . . 175
7.4.2 Najnepovoljniji slučaj . . . 175
7.4.3 Najpovoljniji slučaj . . . 176
7.4.4 Detaljna anliza prosečnog slučaja . . . 178
7.4.5 Izbor pivota . . . 179
7.5 Merge sort . . . 179
7.6 Heap sort . . . 183
7.7 Shell-ovo sortiranje . . . 185
7.8 Sortiranje prebrajanjem . . . 188
7.9 Radix sort . . . 188

8 Grafovi **191**
8.1 Pojmovi i oznake . . . 191
8.2 Reprezentacija grafa . . . 193
8.3 Obilazak grafa . . . 198
8.3.1 Obilazak u dubinu . . . 199
8.3.2 Jedna modifikacija obilaska u dubinu . . . 200
8.3.3 Klasifikacija ivica grafa . . . 203
8.3.4 Određivanje artikulacionih tačaka grafa . . . 204
8.3.5 Mostovi grafa . . . 207
8.3.6 Topološko sortiranje grafa . . . 208
8.3.7 Jako (strogo) povezane komponente . . . 212

8.3.8 Tardžanov algoritam za određivanje jako povezanih komponenti . . . 216
8.4 Najkraća rastojanja u grafu . . . 219
8.4.1 Dijkstrin algoritam . . . 220
8.4.2 Belman–Fordov algoritam . . . 225
8.4.3 Najkraća rastojanja između svih parova čvorova . . . 228
8.5 Minimalno stablo razapinjanja . . . 229
8.5.1 Primov algoritam . . . 230
8.5.2 Korektnost Primovog algoritma . . . 233
8.5.3 Kruskalov algoritam . . . 233

9 Pretraživanje **239**
9.1 Uvodne napomene o pretraživanju . . . 239
9.2 Pretraživanje po uređenom nizu . . . 240
9.2.1 Binarno pretraživanje . . . 242
9.2.2 Interpolirano pretraživanje . . . 243
9.2.3 Fibonačijevo pretraživanje . . . 245
9.2.4 Umetanje nove vrednosti . . . 246
9.2.5 Brisanje postojeće vrednosti . . . 247
9.2.6 Određivanje broja ključeva iz zadatog opsega . . . 247
9.3 Pretraživanje pomoću binarnih stabala . . . 248
9.3.1 Implementacija metode `rank` . . . 249
9.3.2 Implementacija metode `ceil` . . . 250
9.3.3 Implementacija metode za umetanje novog para (ključ, vrednost) . . . 251
9.3.4 Implementacija metode za brisanje para (ključ, vrednost) . . . 251

10 Heš tabele **253**
10.1 Tabele sa direktnim adresiranjem . . . 253
10.2 Heš tabele . . . 253
10.2.1 Razrešavanje kolizija ulančavanjem . . . 254
10.2.2 Analiza složenosti heširanja sa ulančavanjem . . . 255
10.2.3 Otvoreno adresiranje . . . 256
10.2.4 Moguće variajnte za računanje sekvence za provere kod otvorenog adresiranja . . . 258
10.2.5 Složenost osnovnih operacija nad heš tabelom sa otvorenim adresiranjem . . . 260
10.3 Postupci za izračunavanje heš funkcije . . . 262

Predgovor

Ova knjiga je pisana kao udžbenik za predmet *Algoritmi i strukture podataka*, koji se sluša u trećem semestru nastave na Računarskom fakultetu. Knjiga je nastala na osnovu iskustva stečenog tokom izvođenja nastave školske 2015/2016, 2016/2017. i 2017/2108. godine.

Većina izloženog materijala je praćena ilustracijama i implementacijama odgovarajućih struktura i algoritama u programskom jeziku *JAVA*. Razlog za to je što se programski jezik *JAVA* detaljno izučava u drugom semestu nastave i pruža mogućnost za dosta pregledno i čitljivo zapisivanje rešenja.

Na veoma pažljivom čitanju i brojnim korisnim savetima zahvaljujem recenzentima dr Filipu Mariću, vanrednom profesoru Matematičkog fakulteta, Univerziteta u Beogradu i dr Stevanu Milinkviću, redovnom profesoru Računarskog Fakulteta Union Univerziteta. Oni su dali niz korisnih sugestija, koje su svakako doprinele kvalitetu izloženog materijala.

Glava 1

O algoritmima i složenosti

1.1 Pojam algoritma

Pojam algoritma se obično prilično neformalno definiše (mada svakako postoje i vrlo precizne/stroge definicije). Neformalno, *algoritam* je strogo definisana procedura izračunavanja (u originalu *computational procedure*) koja dobija neki podatak ili skup (kolekciju) podataka kao *ulaz* i izračunava (proizvodi) jedan ili više podataka koji predstavljaju *izlaz* ili *rezultat* algoritma. Možemo reći da je algoritam strogo definisan niz koraka (radnji) koje treba izvršiti da bi se od ulaznih podataka dobili izlazni podaci (rezultati). Ovde je naglasak na *strogo definisan* i to treba da podrazumeva da taj niz koraka ne može biti interpretiran na više različitih načina, već samo na jedan način. Pored toga ostaje mala nedoumica šta su zapravo koraci (u frazi *niz koraka*).

Neki autori kažu da se algoritam može posmatrati i kao *alat* za rešavanje dobro specificiranih (definisanih) računskih (u originalu *computational*) problema. Specifikacija problema nije ništa drugo nego postavka problema. Postavka problema može predstavljati opis veze (relacije) koja postoji između ulaznih i izlaznih podataka tog problema. Algoritam opisuje proceduru (tj. niz koraka) koje treba izvesti kako bi se dostigla ta veza između ulaznih i izlaznih podataka.

Na primer, manje više svima je prilično jasno šta predstavlja problem sortiranja. To je jedan od problema koji se najčeše rešava (obično kao deo nekog složenijeg problema). Evo kako bi mogla da izgleda postavka problema sortiranja, u duhu zadavanja veze između ulaza i izlaza:
Ulaz: Niz od n brojeva $\langle a_1, a_2, a_3, ..., a_n \rangle$.
Izlaz: Permutacija (preuređivanje) $\langle a'_1, a'_2, a'_3, ..., a'_n \rangle$ elemenata ulaznog niza tako da je $a'_1 \leqslant a'_2 \leqslant a'_3 \leqslant \cdots \leqslant a'_n$.

Par napomena u vezi sa gornjom postavkom. Indeksiranje elemenata u ovoj postavci kreće od 1, ali isto tako može krenuti i od nule (kao što je u velikom broju programskih jezika koji su trenutno najzastupljeniji). Elementi niza ne moraju biti brojevi, već bilo kakvi podaci koji mogu da se porede (tj. može da se odgovori na pitanje da li su jednaki ili ne, a ako nisu jednaki, koji od njih je veći).

Izlaz, tj. rezultat izvršavanja algoritma za sortiranje niza može biti i permutacija

$$\pi(1), \pi(2), \pi(3), ..., \pi(n)$$

skupa $\{1, 2, 3, ..., n\}$, sa osobinom da je

$$a_{\pi(1)} \leqslant a_{\pi(2)} \leqslant a_{\pi(3)} \leqslant \cdots \leqslant a_{\pi(n)}.$$

Odnosno, broj $\pi(1)$ predstavlja indeks najmanjeg elementa niza a, broj $\pi(2)$ idneks drugog najmanjeg elementa niza a, itd., broj $\pi(n)$ indeks najvećege elementa niza a.

Vratimo se na problem sortiranja. U skladu sa opisanom vezom između ulaza i izlaza, ako je ulazni niz bio $\langle 31, 43, 59, 24, 47, 56\rangle$, onda bi algoritam sortiranja trebao kao izlazni podatak da odredi niz $\langle 24, 31, 43, 47, 56, 59\rangle$. Poslednji niz predstavlja stvarno permutaciju ulaznog niza i to permutaciju u kojoj je svaki element (osim poslednjeg) manji od ili jednak narednom elementu niza. Ulazni niz $\langle 31, 43, 59, 24, 47, 56\rangle$ se naziva *(test) instanca* za problem sortiranja. U opštem slučaju *(test) instanca za neki algoritamski problem* je bilo koja vrednost ili skup (kolekcija) vrednosti koja zadovoljava sva ograničenja za ulaz opisana u postavci problema.

Očigledno da za skoro svaki algoritamski problem možemo napraviti proizvoljno (neograničeno) mnogo različitih test instanci. Algoritam A koji se koristi za rešavanje problema P je *korektan* ako za bilo koju ispravnu (test) instancu određuje ispravan izlaz. Ako postoji bar jedna (test) instanca problema P za koju algoritam A proizvodi pogrešan izlaz, ili se iz nekog razloga njegovo izvršavanje ne završava, algoritam A nije korektan. Provera (utvrđivanje) korektnosti algoritma je nedvosmisleno od velikog značaja. Ono se ne izvodi ponovljenim, izvršavanjem algoritma za različite (test) instance, jer to što algoritam A proizvodi ispravan izlaz za proizvoljno mnogo testiranih instanci ne znači da ne postoji test instanca za koju će proizvesti neispravan izlaz, ili će se izvršavati proizvoljno dugo. Zbog toga se koristi u osnovi matematički aparat (alat) kojim se pokazuje da između ulaznih i izlaznih podataka postoji veza koja je opisana u postavci problema.

Intuitivno je jasno da različite test instance nisu podjednako teške (komplikovane) za neki algoritam. Primera radi za očekivati je da je teže sortirati niz od 1000 elemenata nego niz koji ima 10 elemenata. U ovom slučaju brojevi 1000 i 10 nose određene informacije o test instancama. U literaturi se koristi termin veličina (engl. *size*) kao informacija koja govori o test instanci. U principu to je najčešće neki ceo pozitivan broj koji se pridružuje instanci, ali postupak određivanja tog broja zavisi od problema. Primera radi, za problem sortiranja niza, nameće se da je to broj elemenata u nizu. Ali za problem množenja dva cela broja to može biti broj bitova (cifara), koji se koriste da bi se zapisali ti brojevi. Nekada se veličina izražava i sa više brojeva. Na primer, ako je ulaz za algoritam neki graf (skup čvorova i skup ivica gde ivice povezuju parove čvorova), onda veličinu mogu određivati i broj čvorova i broj ivica.

Evo još par algoritamskih problema koje ćemo opisati, zapisujući vezu između ulaznih i izlaznih podataka. Jedan od njih je problem vraćanja kusura.

Ulaz: Niz brojeva $\langle v_1, v_2, ..., v_n \rangle$ koji predstavljaju vrednosti novčanica u nekoj zemlji i broj I koji predstavlja iznos koji treba kasir da vrati kao kusur.
Izlaz: Niz od n celih nenegativnih brojeva $\langle c_1, c_2, ..., c_n \rangle$ koji predstavljaju brojeve primeraka pojedinih novčanica u kusuru, tako da kusur bude vraćen sa što manje novčanica. Prema tome treba da važi da je

$$c_1 \times v_1 + c_2 \times v_2 + \cdots + c_n \times v_n = I$$

ali i da je

$$c_1 + c_2 + \cdots + c_n$$

što manje.

Problem trgovačkog putnika je jedan od svakako najpoznatijih i najviše proučavanih algoritamskih problema. Intutitivno, trgovački putnik treba da prođe kroz nekih n mesta, prolazeći kroz svako od njih tačno jednom i da se vrati u mesto iz koga je krenuo, ali tako da pređe najkraći put. Formalno, veza između ulaza i izlaza bi se mogla zapisati na sledeći način:
Ulaz: Prirodan broj n koji predstavlja broj mesta (smatramo da su mesta numerisana brojevima od 1 do n) i matrica d dimenzija $n \times n$, tako da je d_{ij} dužina puta između mesta i i j.
Izlaz: Permurtacija π brojeva od 1 do n koja određuje redosled u kome je trgovački putnik obilazio mesta. Ta permutacija treba da bude takva da je

$$\sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}$$

minimalno.

1.2 Zapisivanje algoritama

Algoritmi se mogu prikazivati na razne načine. Verovatno je najstariji način prikazivanja pomoću algoritamskih šema (ili *dijagrama toka*). U ovoj varijanti se algoritam prikazuje grafički (vizuelno) pomoću elemenata koji predstavljaju različite vrste obrada:

- pravougaonici predstavljaju blok koji sadrži neko izračunavanje;
- šestouglovi (rombovi) predstavljaju blok u kome se vrše neka poređenja (testiranja ili ispitivanje) i na osnovu rezultata poređenja (testiranja ili ispitivanja) izvršavanje produžava u jednom od više raznih pravaca;
- blokovi u obliku trapeza služe za učitavanje podataka, ili prikazivanje rezultata.

Ovo su samo neki od mogućih elemenata. Elementi su povezani usmerenim dužima, gde usmerenje određuje kojim redosledom se izvršavaju radnje prikazane u tim blokovima. Ovo je vrlo pregledan način za prikazivanje algoritama, ali nije praktičan

zato što zahteva puno prostora i za zapis obimnijih algoritama će biti potrebno po nekoliko strana, što otežava praćenje algoritma.

Drugi način je tekstualni. U ovom slučaju se algoritam organizuje kao niz koraka i koraci se numerišu brojevima od 1 pa nadalje. Svaki korak se sastoji od opisa radnje koja se izvodi i na kraju opisa zadaje koji je korak sledeći (npr. ako je $a > b$ idete na Korak 5, u suprotnom idete na Korak 8). Nepraktičnost ovakvog načina zapisivanja je što ne mora biti baš u potpunosti precizan, a kao što smo rekli algoritam je niz strogo (precizno) definiasnih koraka.

Treći način je takozvani pseudokod ili pseudojezik. Svaki pseudojezik je „podskup" nekog programskog jezika koji je dovoljan da precizno iskaže algoritam, a istovremeno se „preskaču" ili izostavljaju koraci koji nisu suštinski bitni za algoritam. Na taj način algoritam se skraćuje i pojednostavljuje se praćenje i analiza. Nedostatak je što ne može da se izvrši na računaru i na taj način bar delimično proveri njegova ispravnost.

Četvrti način je korišćenje nekog konkretnog programskog jezika. Time se postiže mogućnost provere osmišljenog algoritma, ali se zato produžava zapis i otežava čitanje/praćenje.

Mi ćemo u ovoj knjizi kombinovati treći i četvrti način. Pri tome ćemo koristiti programski jezik `JAVA`.

1.3 Složenost algoritma

Pri rešavanju algoritamskih problema, osnovni cilj je razviti korektan algoritam, tj. algoritam koji će za svaku ispravnu test instancu određivati tačne izlazne rezultate. Međutim, osim tog osnovnog cilja postoje i drugi ciljevi. Jedan od najvažnijih ciljeva je da se razvije što efikasniji algoritam. Efikasnost je karakteristika algoritma koja govori o tome koliko resursa računara koristi taj algoritam. Kada se govori o resursima pre svega se misli na vreme, ali vrlo često i na količinu upotrebljene memorije.

Vreme može podrazumevati efektivno vreme izvršavanja algoritma, ali je mnogo praktičnije (zbog razlika u brzinama računara na kojima se algoritmi izvršavaju) izražavati procenom broja izvršenih elementarnih koraka u toku izvršavanja algoritma. Pri tome se broj elementarnih koraka određuje u funkciji od veličine instance algoritamskog problema.

U proceni se, radi pojednostavljenja, polazi od određenih pretpostavki. Pretpostavka je da se algoritam izvršava na jednoprocesorskom računaru (sa samo jednim jezgrom). Preciznije, pretpostavka je da se koraci algoritma izvršavaju jedan za drugim, tj. naredni korak se izvršava tek po okončanju prethodnog koraka. Drugim rečima, u jednom vremenskom trenutku ne može biti izvršavano više koraka algoritma, iako su oni takvi po prirodi da bi se mogli izvršavati u isto vreme (ne zavise jedan od drugog, rezultati jednog od njih se ne koriste kao ulazni podaci za drugi). Takođe se pretpostavlja da sve elementarne računske operacije (sabiranje, oduzimanje, množenje, deljenje, poređenje brojeva), kao i rukovanje (baratanje) podacima (pristup promenljivoj ili elementu niza, kopiranje vrednosti

promenljive, itd.) imaju isto vreme izvršavanja.

1.3.1 Analiza složenosti metode za računanje minimuma niza

Ilustrujmo na nekoliko primera određivanje složenosti algoritma. Započnimo sa programom koji određuje minimalni element nekog niza. Jedna od mogućih implementacija metode koja određuje minimum niza sastavljenog od elemenata klase T, koji se mogu porediti prikazan je u listingu 1.1.

Listing 1.1 : Jedna implementacija metode za određivanje minimuma niza.

```
public static <T extends Comparable<T>> T min(T []niz) {
  if (niz.length == 0) return null;
  T res = niz[0];
  for (int k = 1; k < niz.length; k++)
    if (niz[k].compareTo(res) < 0)
      res = niz[k];
  return res;
}
```

Da bi se odredila ukupna složenost metode `min`, potrebno je odrediti složenost pojedinih delova ove metode. Prvi red se sastoji od poređenja dužine niza i broja nula i složenost tog koraka je jednaka jedan. Ako je uslov u poređenju bio ispunjen, prekida se izvršavanje metode (ali, jasno to se dešava samo kada je dužina niza, tj. veličina rešavanog problema bila jednaka nuli). Drugi red predstavlja dodelu vrednosti i njegova složenost je 1. Treći red predstavlja početni deo petlje. Kao što je poznato, blok u zagradama se sastoji od tri dela, razdvojena znakom tačka i zapeta: inicijalizacija (koja se izvršava samo jednom), provera uslova (koji se izvršava sve do prvog izračunavanja u kome je uslov netačan) i deo koji se izvršava nakon izvršavanja tela petlje, a izvršava se svaki put kada je uslov ispunjen, tj. kada se izvrši i telo petlje. Uslov se testira za vrednosti $k = 1, 2, ..., n$ (gde je n broj elemenata niza), pa je broj izračunavanja uslova n. Broj izvršavanja trećeg bloka je $n-1$ (za vrednosti $k = 1, 2, ..., n-1$). Telo petlje se ponavlja $n-1$ puta, a sastoji se od poređenja jednog elementa niza sa trenutnim minimalnim elementom niza i ažuriranja vrednosti trenutnog minimuma (ako je vrednost poređenja `true`). Poređenje će se svakako izvršiti pri svakom izvršavanjau tela petlje, a dodela u nekim situacijama (o tome će biti više reči nešto kasnije). Označimo sa t_n broj izvršavanja te dodele (tako možemo reći da je $0 \leqslant t_n \leqslant n-1$). Konačno, na kraju tela metode je red u kome se vrši povratak iz metode, kao i vraćanje odgovarajućeg rezultata. Znači, ukupna složenost metode `min` će biti:

$$T(n) = 1 + 1 + 1 + n + (n-1) + (n-1) + t_n + 1 = 3n + 2 + t_n. \tag{1.1}$$

Kao što je već rečeno, vrednost izraza t_n je između 0 i $n-1$, pa ako se zamene u gornjem izrazu najmanja i najveća vrednost, dobiće se najmanja moguća i najveća moguća vrednost za složenost metode:

$$T_{\min}(n) = 3n + 2, \quad T_{\max}(n) = 4n + 1. \tag{1.2}$$

Postavlja se pitanje, koliko bi se puta za neki prosečni niz (niz formiran nasumičnim razmeštajem njegovih elemenata) izvršilo ažuriranje vrednost promenljive `res`. Da bi se to procenilo, definišu se slučajne promenljive X_k $(k = 1, 2, ..., n-1)$, na sledeći način:

$$X_k = \begin{cases} 1, & k\text{-ti element niza je bio manji od trenutnog minimuma} \\ 0, & k\text{-ti element niza nije bio manji od trenutnog minimuma} \end{cases} \tag{1.3}$$

Kada kažemo trenutnog minimuma, mislimo na vrednost promenljive `res` u trenutku kada je vrednost „brojačke promenljive" u `for` petlji baš jednaka k. Drugim rečima, trenutni minimum je minimum dela niza od niz_0 do niz_{k-1}. Jasno, u k-tom prolazu kroz petlju (tj. u onom prolazu u kome se obrađuje k-element niza), izvršeno je ažuriranje trenutnog minimuma samo ako je vrednost slučajne promenljive X_k jednaka 1. Matematičko očekivanje promenljive X_k je jednako prosečnom broju izvršavanja dodele `res = niz[k]` (malo nelogično da to može biti broj koji nije ceo, ali to možemo tretirati kao prosečni broj u više ponovljenih izvršavanja metode, a kako je prosečni broj količnik, on može biti i razlomljen). Ukupan broj izvršenih ažuriranja trenutnog minimuma je zbir matematičkih očekivanja za slučajne promenljive X_k $(k = 1, 2, 3, ..., n-1)$.

Očekivanje slučajne promenljive X_k je jednako verovatnoći da je k-ti element niza manji od trenutnog minimuma, tj. verovatnoći da je k-ti element niza manji od minimuma elemenata niza pre tog k-tog elementa. Znači to je verovatnoća da je k-ti element niza manji od svih elemenata pre tog k-tog elementa. Po klasičnoj teoriji verovatnoće, to je jednako količniku broja povoljnih slučajeva i broja mogućih slučajeva. Mogući slučajevi su svi mogući rasporedi elemenata niza `niz[0], niz[1], ..., niz[k-1], niz[k]` i njih ima $(k+1)!$. Povoljni su oni slučajevi kada je `niz[k]` manji od svih ostalih. Prema tome, povoljni su oni rasporedi kod kojih je na mestu `niz[k]` najmanji od elemenata, dok su ostali elementi raspoređeni proizvoljno na mestima `niz[0], niz[1], ..., niz[k-1]`. Takvih rasporeda ima $k!$. Prema tome

$$P(X_k = 1) = \frac{k!}{(k+1)!} = \frac{1}{k+1}, \quad P(X_k = 0) = 1 - P(X_k = 1) = \frac{k}{k+1}, \tag{1.4}$$

pa je očekivanje slučajne promenljive X_k jednako

$$E(X_k) = 0 \cdot P(X_k = 0) + 1 \cdot P(X_k = 1) = \frac{1}{k+1}. \tag{1.5}$$

Zbir matematčkih očekivanja je

$$\sum_{k=1}^{n-1} E(X_k) = \sum_{k=1}^{n-1} \frac{1}{k+1} \tag{1.6}$$

Konačno, poslednji izraz se može ograničiti sa obe strane

$$\int_2^n \frac{1}{x} dx < \sum_{k=1}^{n-1} \frac{1}{k+1} < \int_1^{n-1} \frac{1}{x} dx \tag{1.7}$$

odnosno

$$\ln \frac{n}{2} < \sum_{k=1}^{n-1} \frac{1}{k+1} < \ln(n-1) < \ln n. \tag{1.8}$$

Prema tome, prosečan broj ažuriranja promenljive koja sadrži minimalni element niza tokom izvršavanja „standardnog" algoritma za određivanje minimuma niza je između $\ln \frac{n}{2}$ i $\ln n$.

1.3.2 Analiza složenosti sortiranja niza primenom sortiranja selekcijom

Drugi primer na kome će biti ilustrovano računanje broja elementarnih koraka tokom izvršavanja algoritma je sortiranje elemenata niza. U listingu 1.2 je prikazana jedna od implementacija za sortiranje niza postupkom poznatim pod nazivom sortiranje selekcijom (*selection sort*). Kao što se iz listinga vidi, pretpostavka je da su elementi niza nekog tipa `T`, koji implementira interfejs `Comparable`, što znači da se elementi niza mogu porediti pozivanjem metode `compareTo`. Pretpostavka je da niz koji se sortira ima n elemenata.

Listing 1.2 : Jedna implementacija metode za sortiranje selekcijom.

```
public static <T extends Comparable<T>>
        void selectionSort(T []niz) {
  for (int i = 0; i < niz.length; i++) {
    k = i;
    for (int j = i+1; j < niz.length; j++)
      if (niz[j].compareTo(niz[k]) < 0) k = j;
    DSUtil.swap(niz, i, k);
  }
}
```

Prvi red metode predstavlja početak `for` petlje. U toku izvršavanja te petlje ukupno $n+1$ puta će biti testiran uslov (za vrednosti $i = 0, 1, 2, ..., n$) a n puta će vrednost promenljive `i` biti uvećavana. U svakom prolazu kroz spoljašnju petlju će biti inicijalizovana vrednost promenljive k (pa će biti n tih inicijalizacija) i biće pozvana metoda `swap` koja vrši razmenu vrednosti dva elementa niza (te je složenost tog dela $3n$, ako ignorišemo poziv metode i povratak). U nekom konkretnom izvršavanju spoljašnje petlje, unutrašnja petlja se izvršava $n - i - 1$ puta (za vrednosti $j = i + 1, i + 2, ..., n - 1$) i toliko puta se promenljiva j uvećava za 1, a za jedan put više testira uslov za nastavak izvršavanja unutrašnje petlje. Inicijalizacija vrednosti promenljive j (unutrašnja petlja) ima složenost 2 (sabiranje i upis u prostor predviđen za promenljivu j). Telo unutrašnje petlje se sastoji od izračunavanja uslova (poređenje dva elementa niza), koji se svakako izvršava i ažiriranja vrednosti promenljive k, koje se izvršava samo ako je uslov ispunjen (neka je t_i broj izvršavanja te dodele u i-tom izvršavanjau spoljašnje petlje). Konačno se može zapisati složenost kompletne metode

$$T(n) = 1 + (n+1) + n + n + 3n +$$