Algoritmi ``Podijeli pa vladaj''

Penzer, Petra

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Petra Penzer

ALGORITMI ”PODIJELI PA VLADAJ”

Diplomski rad

Voditelj rada:
izv.prof.dr.sc. Saša Singer

Zagreb, rujan 2016.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Prilikom oblikovanja algoritama za rješavanje raznih problema koristimo se jednom od klasičnih metoda dizajniranja algoritama. Najpoznatije metode su: ”podijeli pa vladaj”, dinamičko programiranje, pohlepni pristup i strategija odustajanja.

U ovom diplomskom radu će biti detaljnije opisana metoda ”podijeli pa vladaj” te nekoliko primjera algoritama koji su oblikovani ovom metodom. Vidjet ćemo da je ova metoda vrlo djelotvorna u mnogim situacijama.

Poglavlje 1

Metoda ”podijeli pa vladaj”

Metoda ”podijeli pa vladaj” je jedna od najprimjenjivijih tehnika za oblikovanje algoritama. Koristi se za rješavanje raznih problema kao što su pretraživanje te sortiranje nizova, traženje najbližeg para točaka u ravnini, množenje velikih cijelih brojeva, množenje matrica, traženje k-tog najmanjeg elementa u nizu brojeva itd. U idućem poglavlju ću opisati algoritme za rješavanje upravo nekih od ovih problema.

1.1 Koraci algoritama

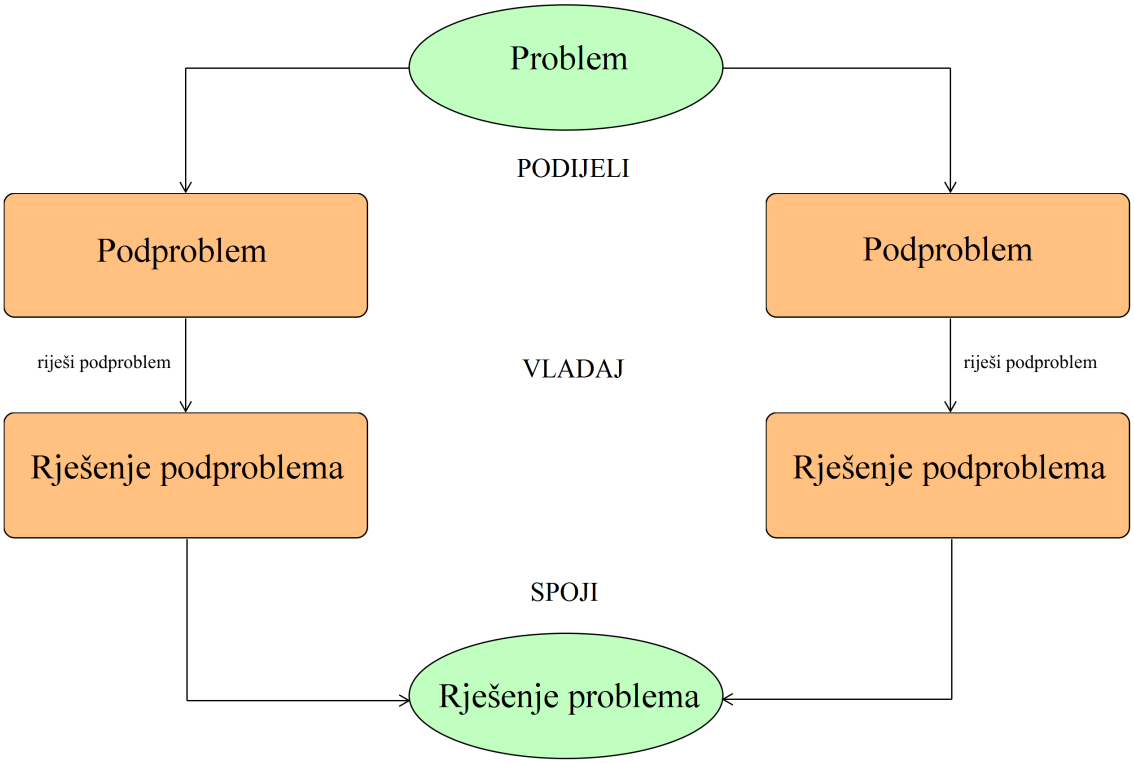
Algoritmi oblikovani spomenutom metodom sastoje se od sljedećih koraka:

1. korak podjele
2. korak vladanja
3. korak spajanja

Slika 1.1 ilustrira korake algoritama oblikovanih metodom ”podijeli pa vladaj”. Sada opišimo detaljnije što se događa u svakom od navedenih koraka algoritma.

Korak podjele

Na početku svakog algortima početni problem veličine $n \in \mathbb{N}$ dijelimo na $p \geq 2$ podproblema podjednake veličine. Podproblemi su isti problemi kao početni problem, ali manje veličine. Vrijednost cijelog broja p je najčešće 2, no i ostale vrijednosti veće od 2 su česte. Uočimo da slika 1.1 prikazuje podjelu početnog problema upravo na 2 podproblema.



Slika 1.1: Koraci algoritama

Korak vladanja

Nakon što smo u prošlom koraku podijelili početni problem na p podproblema, u ovom koraku rekurzivno rješavamo neke ili svaki od tih podproblema. Ukoliko je zadovoljen određeni uvjet u promatranom algoritmu, tada podprobleme rješavamo direktno. Inače, ako taj uvjet nije zadovoljen, tada podproblem rješavamo rekurzivno, pozivanjem početnog algoritma na tom podproblemu. Na primjer, u idućem poglavlju ćemo vidjeti da je uvjet kod *mergesort* algoritma zadovoljen ukoliko je duljina promatranog niza jednaka 1, a nije zadovoljen ukoliko je duljina niza veća od 1.

Korak spajanja

Nakon što smo u prethodnom koraku riješili svih p podproblema, ovaj korak ta rješenja spaja na određeni način, tako da dobijemo željeni izlaz koji je rješenje početnog problema veličine n. Na primjer, u algoritmu *mergesort*, kao što ćemo vidjeti u idućem poglavlju, na početku ovog koraka imamo dvije sortirane liste te ih funkcijom *merge* spajamo u jednu sortiranu listu.

Korak spajanja se u "podijeli pa vladaj" algorimima može sastojati od spajanja, sortiranja, traženja, itd. Efikasnost algoritama ovisi o tome koliko promišljeno se provodi upravo ovaj korak algoritma.

Dakle, svi "podijeli pa vladaj" algoritmi se sastoje od sljedećih koraka, točno tim redoslijedom:

1. Ako je veličina početnog problema I dovoljno mala, odnosno, ukoliko je zadovoljen određeni uvjet, tada riješi problem direktno i vrati odgovor.
2. Inače, podijeli problem I na p podproblema $I_1, I_2, \ldots, I_p$ približno iste veličine.
3. Rekurzivno pozovi algoritam na svakom podproblemu I_j, $1 \leq j \leq p$, tako da dobiješ p djelomičnih rješenja.
4. Spoji p djelomičnih rješenja tako da dobiješ rješenje originalnog problema I.

1.2 Tipovi algoritama "podijeli pa vladaj"

Postoje dva podtipa algoritama oblikovanih metodom "podijeli pa vladaj", ovisno o tome na koliko se podproblema rekurzivno zove algoritam.

1. Prvi podtip dijeli početni problem na p podproblema podjednake veličine i $p - 1$ podproblema u potpunosti zaboravlja, jer se u njima ne nalazi traženo rješenje, te rekurzivno poziva algoritam samo na jednom podproblemu. Primjer ovog podtipa su: binarno pretraživanje, hanojski tornjevi, pronalaženje minimalnog odnosno maksimalnog elementa u listi itd.
2. Drugi podtip dijeli početni problem na p podproblema podjednake veličine te na svakom podproblemu poziva rekurzivno početni algoritam. Primjer ovog podtipa su: $mergsort$ algoritam, $quicksort$ algoritam, itd.

1.3 Složenost algoritama

Prilikom oblikovanja algoritama cilj je oblikovati što učinkovitiji algoritam. Jedan od načina mjerenja učinkovitosti algoritama je računanje vremenske složenosti i upravo taj kriterij ću koristiti za računanje složenosti algoritama u ovom diplomskom radu.

Vremenska složenost algoritma je vrijeme potrebno za izvođenje algoritama i izražavamo je matematičkom funkcijom.

Jedan od načina procjene složenosti algoritama je O-notacija, stoga je definirajmo formalno (vidi [6], str. 18), jer ćemo je često koristiti u sljedećem poglavlju.

Definicija 1.3.1. *Neka su $f, g : \mathbb{N} \to \mathbb{R}^+$ dvije funkcije. Kažemo da je funkcija g asimptotska gornja međa za funkciju f ako postoje $c > 0$ i $n_0 \in \mathbb{N}$ takvi da za svaki $n \geq n_0$ vrijedi $f(n) \leq c \cdot g(n)$. Oznaka: $f(n) = O(g(n))$.*

1.4 Rekurzivne relacije

Prilikom analize složenosti algoritama često se susrećemo s homogenim i nehomogenim rekurzivnim relacijama koje detaljnije opisujem prema [8].

Homogene rekurzivne relacije

Linearne homogene rekurzivne relacije s konstantnim koeficijentima imaju oblik:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0, \quad 1 \leq k \leq n, \tag{1.1}$$

gdje su koeficijenti a_i konstante i $a_0 \neq 0$.

Relacije ovog oblika rješavamo na sljedeći način:

1. Uvodimo supstituciju $t_n := x^n$ te dobivamo jednadžbu:
$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = 0. \tag{1.2}$$
2. Jednadžbu (1.2) podijelimo s x^{n-k} i dobijemo jednadžbu:
$$a_0 x^k + a_1 x^{k-1} + \cdots + a_k = 0. \tag{1.3}$$
koju zovemo karakteristična jednadžba rekurzivne relacije (1.1) i ona ima točno k rješenja.

Ovisno o rješenjima karakteristične jednadžbe razlikujemo dva opća rješenja relacije (1.1):

1. Ako su $x_1, x_2, \ldots, x_k$ različita rješenja karakteristične jednadžbe tada je
$$a_n = c_1 x_1^n + c_2 x_2^n + \cdots + c_k x_k^n \tag{1.4}$$
opće rješenje relacije (1.1), gdje su c_i konstante.
2. Ako rješenja $x_1, x_2, \ldots, x_k$ nisu međusobno različita, neka su $x_1, x_2, \ldots, x_r, (1 \leq r \leq k)$ različita rješenja karakteristične jednadžbe, takva da je kratnost od x_i jednaka m_i, za $i = 1, \ldots, r$. U tom slučaju je opće rješenje relacije (1.1):
$$\begin{aligned} a_n = (c_{11} + c_{12} n + c_{13} n^2 + \cdots + c_{1m_1} n^{m_1 - 1}) x_1^n + \cdots \\ + (c_{r1} + c_{r2} n + c_{r3} n^2 + \cdots + c_{rm_r} n^{m_r - 1}) x_r^n, \end{aligned}$$
gdje su c_{ij} konstante.

Konstante c_i, odnosno, c_{ij} određujemo iz početnih uvjeta.

Nehomogene rekurzivne relacije

Linearne nehomogene rekurzivne relacije s konstantnim koeficijentima imaju oblik:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = g(n), \quad 1 \leq k \leq n, \tag{1.5}$$

gdje su koeficijenti a_i konstante i $a_0 \neq 0$.

Način rješavanja nehomogenih rekurzivnih relacija ovisi o funkciji $g(n)$:

1. Ako je $g(n) = b^n p_d(n)$, $n \geq k$, $b \in \mathbb{C}$, $b \neq 0$ konstanta i p_d polinom stupnja d u varijabli n, tada se relacija (1.5) može pretvoriti u linearnu homogenu rekurziju reda $k + d + 1$ s karakterističnom jednadžbom:

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b)^{d+1} = 0. \tag{1.6}$$

Napomena 1.4.1. *Prethodna tvrdnja vrijedi samo ako konstanta b nije rješenje homogenog dijela jednadžbe (1.6).*

2. Neka je funkcija $g(n)$ suma članova koji imaju oblik funkcije g iz prethodnog primjera, odnosno, funkcija $g(n)$ je:

$$g(n) = b_1^n p_{d_1}(n) + b_2^n p_{d_2}(n) + \cdots + b_l^n p_{d_l}(n), \tag{1.7}$$

pri čemu su konstante b_i međusobno različite, a p_{d_i} su polinomi u n stupnja točno d_i, $i = 1, \ldots, l$. Ovakva rekurzija se, također, može homogenizirati na isti način kao u prvom primjeru. Dobivena homogenizirana rekurzija je reda $k + (d_1 + 1) + \cdots + (d_l + 1)$, s karakterističnom jednadžbom:

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b_1)^{d_1+1} \ldots (x - b_l)^{d_l+1} = 0. \tag{1.8}$$

1.5 Rješavanje bezuvjetnih rekurzivnih relacija

Prilikom rješavanja rekurzivnih relacija u idućem poglavlju, ponekad ću, radi jednostavnosti, uvesti uvjet da je varijabla n potencija nekog cijelog broja $b \geq 2$. Za rješavanje istih rekurzivnih rekurzija bezuvjetno, koristiti ću definicije i propoziciju koje su navedene u nastavku poglavlja (vidi [8], str. 57-58).

Definicija 1.5.1. *Neka je $f : D \to \mathbb{R}_0^+$ nenegativna funkcija na odozgo neograničenom podskupu $D \subseteq \mathbb{R}_0^+$. Funkcija f je asimptotski rastuća ako je f rastuća za dovoljno velike argumente, tj.*

$$\exists M \in D \text{ takav da za svaki } x, y \in D \text{ vrijedi:}$$

$$x, y \geq M \text{ i } x < y \Rightarrow f(x) \leq f(y).$$

Definicija 1.5.2. *Neka je* $f : D \to \mathbb{R}_0^+$ *asimptotski rastuća funkcija i neka je* $b > 1$. *Funkcija* f *je b-glatka ako vrijedi:*

$$f(bx) = O(f(x)).$$

Propozicija 1.5.3. *Neka je* $f : D \to \mathbb{R}_0^+$ *glatka funkcija. Neka je* $b \in \mathbb{N}$, $b \geq 2$ *i* $T : \mathbb{N}_0 \to \mathbb{R}_0^+$ *asimptotski rastuća funkcija, takva da vrijedi uvjetna asimptotska relacija:* $T(n) = O(f(n))$, *n je potencija od b. Tada vrijedi i bezuvjetna asimptotska relacija:*

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

Poglavlje 2

Primjeri metode ”podijeli pa vladaj”

U ovom poglavlju ću opisati nekoliko primjera algoritama koji su oblikovani metodom ”podijeli pa vladaj”.

2.1 Binarno pretraživanje

Jedan od poznatijih primjera metode ”podijeli pa vladaj” je binarno pretraživanje, tj. pretraživanje ”raspolavljanjem”. Binarno pretraživanje koristimo za traženje zadanog elementa x u uzlazno (silazno) sortiranom nizu A, proizvoljne duljine n.

Pretpostavimo da je niz A sortiran uzlazno. Na početku rada algoritma, odredimo srednji element ulaznog niza. Zatim, vrijednost tog srednjeg elementa uspoređujemo s vrijednošću traženog elementa x. Ako je traženi element jednak srednjem elementu, pretraživanje je završeno. Inače, ako je traženi element manji od vrijednosti srednjeg elementa, tada se traženi element x može nalaziti samo u lijevoj polovici niza, stoga desnu polovicu ulaznog niza više ne pretražujemo. Analogno, ako je traženi element veći od vrijednosti srednjeg elementa niza, onda se traženi element x može nalaziti samo u desnoj polovici niza.

Ovaj postupak ponavljamo rekurzivno na polovici niza u kojoj se element x može nalaziti. Nastavljamo traženje dok se traženi element x ne pronađe ili dok se niz za pretraživanje ne isprazni, što znači da se vrijednost x ne nalazi u zadanom nizu.

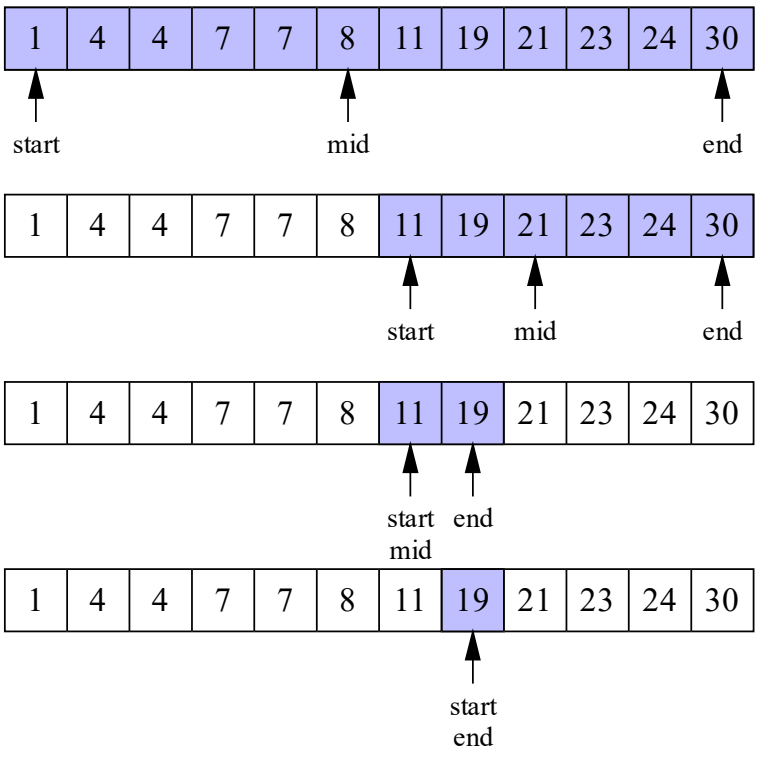
U nastavku je prikazan pseudokod binarnog pretraživanja:

```
BinarySearch(A, start, end, x)
  if (start>end)
    then return -1
  else
```

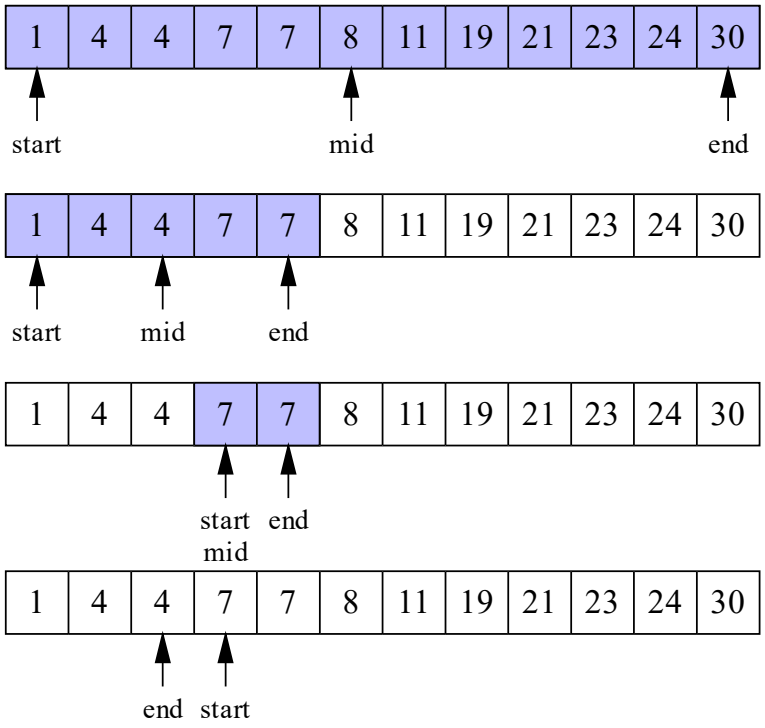
```
mid=(start+end)/2
if (x==A[mid])
  then return mid
else (if x<A[mid])
       then return BinarySearch(A, start, mid-1, x)
     else
       return BinarySearch(A, mid+1, end, x)
```

Algoritam kao parametre prima niz A, indekse *start* i *end* te element x. Indeksi *start* i *end* označavaju dio niza A koji trenutno pretražujemo. Kada indeks *start* postane veći od indeksa *end*, tada u nizu A ne postoji vrijednost x. Ukoliko se element x nalazi u nizu A algoritam vraća njegov indeks u nizu A.

Sljedeće dvije slike (vidi [5]) ilustriraju postupak rada algoritma binarnog pretraživanja na uzlazno sortiranom nizu od 12 elemenata. Slika 2.1 prikazuje traženje elementa $x = 19$ kada se x nalazi u zadanom nizu, a slika 2.2 prikazuje traženje elementa $x = 6$ koji se ne nalazi u zadanom nizu.



Slika 2.1: Prvi primjer rada algoritma binarnog pretraživanja



Slika 2.2: Drugi primjer rada algoritma binarnog pretraživanja

Vremenska složenost binarnog pretraživanja

Da bismo izračunali vrijeme izvršavanja algoritma binarnog pretraživanja potrebno je izračunati broj izvršenih usporedbi elemenata.

Ako je niz prazan algoritam ne izvodi niti jednu usporedbu. Ako niz sadrži samo jedan element tada je izvršena samo jedna usporedba traženog elementa x s tim jednim elementom. Ako je broj elemenata ulaznog niza barem 2 tada imamo 2 mogućnosti. U slučaju da vrijedi $x = A[mid]$, izvršena je samo jedna usporedba. Inače je broj usporedbi potreban za izvršavanje algoritma jednak broju usporedbi učinjenih rekurzivnim pozivom na prvoj ili drugoj polovici niza, uvećanom za jedan.

Označimo s $T_{BINARY}(n)$ broj usporedbi izvršenih na nizu duljine n. Tada vrijedi:

$$T_{BINARY}(n) \leq \begin{cases} 1, & ako\ je\ n = 1 \\ 1 + T_{BINARY}(\lfloor n/2 \rfloor), & ako\ je\ n \geq 2. \end{cases}$$

Raspisivanjem ove rekurzije, za dovoljno velik $k \geq 0$ takav da vrijedi $\lfloor n/2^k \rfloor = 1$,

dobivamo:

$$T_{BINARY}(n) \leq 1 + T_{BINARY}(\lfloor n/2 \rfloor) \leq 2 + T_{BINARY}(\lfloor n/4 \rfloor) \leq$$
$$\leq 3 + T_{BINARY}(\lfloor n/8 \rfloor) \leq \cdots \leq (k-1) + T_{BINARY}(\lfloor n/2^{k-1} \rfloor) = (k-1) + 1 = k.$$

Uočimo da za traženi $k \geq 0$, takav da je $\left\lfloor \frac{n}{2^k} \right\rfloor = 1$, vrijedi:

$$2^{k-1} \leq n < 2^k.$$

Primjenjujući funkciju $\log_2()$ na gornju nejednadžbu dobivamo:

$$k - 1 \leq \log_2 n < k.$$

Odnosno, uvećavanjem za jedan, vrijedi:

$$k \leq \log_2 n + 1 < k + 1.$$

Time dobivamo $k = \lfloor \log_2 n \rfloor + 1$. Konačno, vrijedi nejednakost:

$$T_{BINARY}(n) \leq \lfloor \log_2(n) \rfloor + 1,$$

iz koje slijedi da je vremenska složenost algoritma binarnog pretraživanja $O(\log_2(n))$.

2.2 Mergesort

Mergesort je rekurzivan algoritam koji koristimo za sortiranje nizova. Ovaj algoritam kao ulaz uzima niz brojeva, zatim ga podijeli na dva podniza podjednake duljine. Rekurzivno sortira svaki od tih podnizova i spoji ih u jedan niz početne duljine. Algoritam vraća kao izlaz niz ulaznih brojeva sortiranih silazno ili uzlazno. U nastavku ovog poglavlja opisujem postupak uzlaznog sortiranja niza.

Za početak, opisujem funkciju *merge* koja dva uzlazno sortirana podniza spaja u jedan niz. Promatramo dio niza A od indeksa i do indeksa j, pri čemu vrijedi $i < j$. Neka je indeks m takav da je $i \leq m \leq j$ i neka je prvi podniz dio niza A od indeksa i do m, a drugi podniz od indeksa $m + 1$ do j. Redom uspoređujemo elemente prvog podniza s elementima drugog podniza. Ako je trenutni element prvog podniza manji od trenutno promatranog elementa drugog podniza, tada element prvog podniza kopiramo u pomoćno polje te povećamo indeks u prvom podnizu. Analogno, ako je element drugog podniza manji od elementa prvog podniza, tada kopiramo element drugog podniza u pomoćno polje te povećamo indeks u drugom podnizu. Kada dođemo do kraja jednog od podnizova, tada preostale elemente iz drugog niza do kraja kopiramo u pomoćno polje. Na kraju ove funkcije sve elemente iz pomoćnog polja kopiramo u originalni niz A. Time smo dobili sortirani dio polja A od indeksa i do j.

Dakle, ova funkcija prima kao ulazne parametre indekse i, m i j, te niz A, gdje su $A[i], \ldots, A[m]$ i $A[m + 1], \ldots, A[j]$ uzlazno sortirani podnizovi. Algoritmom se podnizovi spajaju u jedan uzlazno sortiran niz duljine $j - i + 1$.

U nastavku je prikazan pseudokod funkcije *merge*:

```
merge(A,i,m,j) {
  p=i
  q=m+1
  r=i
  while(p<=m && q<=j) {
    if(A[p]<=A[q]) {
      c[r]=A[p]
      p=p+1
    }
    else {
      c[r]=A[q]
      q=q+1
    }
    r=r+1
  }
```

```
   while(p<=m) {
      c[r]=A[p]
      p=p+1
      r=r+1
   }
   while(q<=j) {
      c[r]=A[q]
      q=q+1
      r=r+1
   }
   for r=i to j
      a[r]=c[r]
}
```

Sada opisujem cijeli algoritam *mergesort*. Algoritam *mergesort* prima kao ulaz niz A te indekse i i j, koji označavaju koji dio niza A želimo sortirati trenutnim pozivom algoritma. Na početku algoritma provjeravamo nalazi li se u ulaznom nizu samo jedan element te u tom slučaju ne radimo ništa. Ako ulazni niz sadrži više od jednog elementa, tada odredimo indeks srednjeg elementa u promatranom dijelu niza i time niz podijelimo na dva podniza podjednake duljine te na svakoj polovici pozovemo *mergesort* algoritam. Na kraju spajamo ta dva sortirana podniza u niz početne duljine, koristeći opisanu funkciju *merge*.

Pseudokod *mergesort* algoritma je prikazan u nastavku:

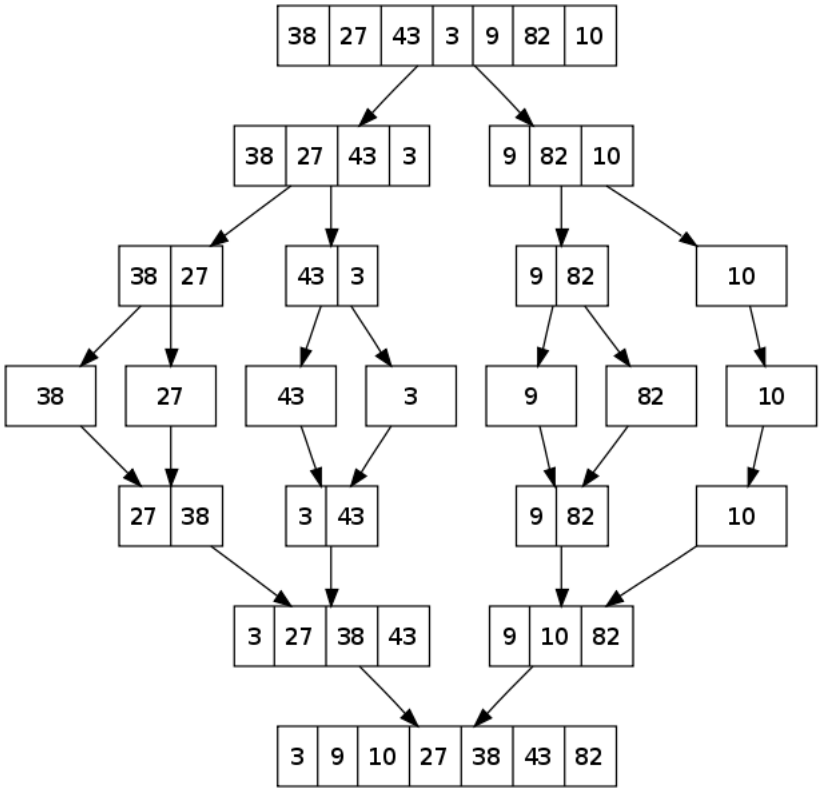
```
mergesort(a,i,j) {
   if(i==j) return
   m=(i+j)/2
   mergesort(a,i,m)
   mergesort(a,m+1,j)
   merge(a,i,m,j)
}
```

Slika 2.3 (vidi [3]) prikazuje primjer rada mergesort algoritma na ulaznom nizu od 7 elemenata.

Vremenska složenost mergesort algoritma

Za računanje vremenske složenosti algortima, potrebno je izračunati broj izvršenih usporedbi i kopiranja elemenata. Ako vrijedi $n = 1$, gdje je n broj elemenata niza, tada je niz već sortiran. Stoga pretpostavimo da je $n > 1$.

Izračunajmo prvo vremensku složenost funkcije *merge*. U svakom prolazu kroz *while* petlju, element niza A je kopiran u element niza c. Dakle, svakom elementu se pristupa



Slika 2.3: Primjer rada mergesort algoritma

točno jednom, stoga zaključujemo kako je vrijeme izvršavanja *while* petlji jednako $O(n)$. Istim zaključivanjem dolazimo do iste vremenske složenosti for petlje. Stoga je vremenska složenost funkcije *merge* jednaka $O(n) + O(n) = O(n)$, gdje je n broj elemenata niza.

Označimo s $T_{MERGESORT}(n)$ broj usporedbi i kopiranja elemenata potrebnih za sortiranje niza duljine n korištenjem *mergesort* algoritma. Odredimo vremensku složenost *mergesort* algoritma u najgorem slučaju. Nakon podjele ulaznog niza na dva podjednaka podniza, njihove duljine su jednake $\lceil n/2 \rceil$ i $\lfloor n/2 \rfloor$. Stoga broj usporedbi elemenata u tim podnizovima označavamo s $T_{MERGESORT}(\lceil n/2 \rceil)$ i $T_{MERGESORT}(\lfloor n/2 \rfloor)$. Budući da smo pokazali kako je vremenska složenost spajanja dva podniza jednaka $O(n)$, dobivamo sljedeću rekurzivnu relaciju za vremensku složenost mergesort algoritma:

$$T_{MERGESORT}(n) = \begin{cases} 0, & ako\ je\ n = 1 \\ T_{MERGESORT}(\lceil n/2 \rceil) + T_{MERGESORT}(\lfloor n/2 \rfloor) + bn, & ako\ je\ n \geq 2, \end{cases}$$

gdje je $b \geq 0$ konstanta.

Riješimo sada ovu rekurzivnu relaciju. Pretpostavljam prvo da je n potencija broja 2. Tada postoji cijeli broj $k \geq 0$ takav da je $n = 2^k$, odnosno, $k = \log_2(n)$. Rekurzija tada ima oblik:

$$T_{MERGESORT}(n) = 2T_{MERGESORT}\left(\frac{n}{2}\right) + bn.$$

Uvrštavanjem supstitucije $n = 2^k$ dobivamo:

$$T_{MERGESORT}(2^k) = 2T_{MERGESORT}(2^{k-1}) + b2^k,$$

odnosno,

$$t_k = 2t_{k-1} + b2^k. \tag{2.1}$$

Koristeći objašnjenja iz prethodnog poglavlja dobivamo da je karakteristična jednadžba jednaka:

$$(x-2)^2 = 0.$$

Stoga je opće rješenje rekurzije:

$$t_k = c_1 2^k + c_2 k 2^k,$$

odnosno,

$$T_{MERGESORT}(n) = c_1 n + c_2 n \log_2 n.$$

Izračunajmo sada konstante c_1 i c_2. Uvrstimo opće rješenje u jednadžbu (2.1):

$$c_1 2^k + c_2 k 2^k = 2(c_1 2^{k-1} + c_2(k-1)2^{k-1}) + b2^k.$$

Sređivanjem prethodne jednadžbe dobivamo: $c_2 = b$.
Iz početnog uvjeta $T_{MERGESORT}(1) = 0$ odredimo konstantu c_1:

$$0 = T_{MERGESORT}(1) = c_1 + b\log_2(1)$$

$$c_1 = 0.$$

Tako dobivamo da je rješenje uvjetne rekurzije:

$$T_{MERGESORT}(n) = bn\log_2(n), \text{ za neku konstantu } b > 0.$$

Očito vrijedi $T_{MERGESORT}(n) \in O(n \log_2 n)$, gdje je n potencija broja 2.

Riješimo sada početnu rekurziju bezuvjetno, koristeći definicije i propoziciju iz prethodnog poglavlja. Potrebno je dokazati sljedeća dva uvjeta:

1. funkcija $T_{MERGESORT}(n)$ je asimptotski rastuća

2. funkcija $f(n) = n \log_2 n$ je glatka funkcija.

Dokaz. Prvi uvjet dokazujem matematičkom indukcijom.

1. Baza indukcije:

$$T_{MERGESORT}(1) = 0 \leq 2T(1) + T(1) + 2b = T(2).$$

Time je zadovoljena baza indukcije.

2. Neka je $n > 1$. Pretpostavimo da za svaki $m \in \mathbb{N}$, takav da je $m < n$, vrijedi $T(m) \leq T(m+1)$. Dokažimo da tada vrijedi $T(n) \leq T(n+1)$.

3. Korak indukcije:

Prvo uočimo sljedeće nejednakosti:

$$\left\lfloor \frac{n+1}{2} \right\rfloor = \begin{cases} \left\lfloor \frac{n}{2} \right\rfloor, & ako\ je\ n\ paran\ broj \\ \left\lfloor \frac{n}{2} \right\rfloor + 1, & ako\ je\ n\ neparan\ broj. \end{cases} \tag{2.2}$$

Analogno,

$$\left\lceil \frac{n+1}{2} \right\rceil = \begin{cases} \left\lceil \frac{n}{2} \right\rceil + 1, & ako\ je\ n\ paran\ broj \\ \left\lceil \frac{n}{2} \right\rceil, & ako\ je\ n\ neparan\ broj. \end{cases} \tag{2.3}$$

Iz prethodnih jednakosti vrijede sljedeće nejednakosti:

$$T_{MERGESORT}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq T_{MERGESORT}\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) \tag{2.4}$$

$$T_{MERGESORT}\left(\left\lceil \frac{n}{2} \right\rceil\right) \leq T_{MERGESORT}\left(\left\lceil \frac{n+1}{2} \right\rceil\right). \tag{2.5}$$

Tada za zadani $n > 1$, koristeći (2.4) i (2.5), vrijedi:

$$T_{MERGESORT}(n) = T_{MERGESORT}(\lceil n/2 \rceil) + T_{MERGESORT}(\lfloor n/2 \rfloor) + bn$$
$$\leq T_{MERGESORT}(\lceil (n+1)/2 \rceil) + T_{MERGESORT}(\lfloor (n+1)/2 \rfloor) + b(n+1) = T(n+1).$$

Ovime je dokazano da je $T_{MERGESORT}(n)$ asimptotski rastuća funkcija.

Sada dokazujem da je funkcija $f(n) = n \log_2 n$ 2-glatka funkcija. Vrijedi za $n \geq 2$:

$$f(2n) = 2n \log_2(2n) = 2n(\log_2 2 + \log_2 n) \leq [\log_2 2 \leq \log_2 n, \forall n \geq 2]$$
$$\leq 2n \log_2 n + 2n \log_2 n = 4n \log_2 n = 4f(n).$$

Time smo dobili $f(2n) \in O(f(n))$, odnosno, funkcija $f(n)$ je 2-glatka. □

Primjenjujući propoziciju 1.5.3 slijedi $T_{MERGESORT}(n) \in O(n \log_2 n)$.

2.3 Quicksort

Quicksort je vrlo popularna rekurzivna metoda sortiranja nizova. Prosječna vremenska složenost ovog algoritma, kao što ćemo kasnije pokazati, je $O(n \log n)$ te se zbog tog u praksi često koristi. Prednost quicksort algoritma nad mergesort algoritmom je sortiranje u mjestu, odnosno, nije potreban pomoćni niz za spremanje sortiranih elemenata.

Ideja quicksort algoritma je sortirati niz $A[1], \ldots, A[n]$ uzlazno tako da odaberemo jedan član niza $A[j], j \in 1, \ldots, n$, kojeg zovemo pivot, te raspodijelimo niz tako da sve članove niza koji imaju vrijednost manju od vrijednosti pivota stavimo lijevo od njega, a sve članove niza koji imaju veću vrijednost od vrijednosti pivota stavimo desno od pivota. Zatim rekurzivno primijenimo quicksort algoritam na lijevu i desnu polovicu niza. Tako ćemo sortirati cijeli početni niz.

Za početak, proučimo problem izbora pivotnog elementa. Taj problem će rješavati funkcija *findpivot*, koja kao ulazne parametre uzima indekse i i j niza A te vraća indeks k odabranog pivota u nizu A, tako da vrijedi $i \leq k \leq j$.

Uočimo, indeks pivotnog elementa u sortiranom nizu, dakle nakon raspodjele elemenata, će biti drugačiji od vrijednosti koju je vratila funkcija *findpivot*. Detaljnije ću to opisati pri opisu funkcije *partition*.

Funkcija *findpivot* prvo provjerava nalaze li se u promatranom dijelu niza dvije različite vrijednosti te veću odabire za pivot. Ukoliko su sve vrijednosti jednake, funkcija ne radi ništa, odnosno, vraća 0.

Pseudokod funkcije *findpivot* je:

```
findpivot(i, j) {
  first = A[i]
  for k=i+1 to j do
    if (A[k]>first) then
      return k
    else if (A[k]<first) then
          return i
  return 0
}
```

Objasnimo sada zašto je bitno da vrijednost pivotnog elementa nije najmanja vrijednost u promatranom dijelu niza, odnosno, zašto od dvije različite vrijednosti biramo veću. Promotrimo sljedeći niz A koji sadrži 6 elemenata:

$$A[6] = \{1, 7, 3, 16, 28, 2\}.$$

Ukoliko broj 1 odaberemo za pivotni element, tada će se svi ostali elementi nakon poziva funkcije *partition* ponovno nalaziti desno od pivota, a lijevo od pivota se neće nalaziti

niti jedan element. Algoritam *quicksort* će pozvati samog sebe na istom dijelu niza i tako ponovno odabrati broj 1 za pivot. Time nastaje beskonačna rekurzija jer se isti posao ponavlja svakim pozivom. U našem rješenju će se, dakle, lijevo od pivota nalaziti barem jedan element, budući da je pivot barem druga najmanja vrijednost.

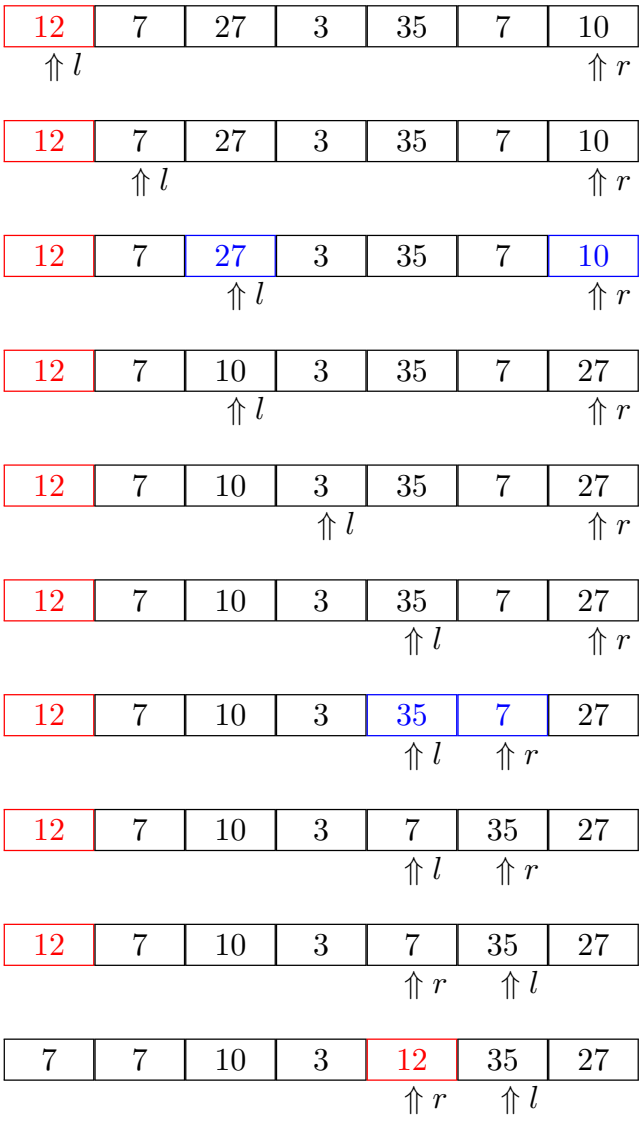
Opišimo sada detaljnije problem raspodjele elemenata oko pivotnog elementa. Ovaj problem rješava funkcija *partition*. Funkcija *partition* prima indekse *i*, *j* i *pivotindex*. Indeksi *i* i *j* određuju koji dio niza *A* trenutno promatramo, a indeks *pivotindex* je indeks pivota koji je vratila funkcija $findpivot$. Uvodimo nove indekse *l* i *r*, koji će označavati lijevu i desnu stranu promatranog niza. Sve elemente niza *A* uspoređujemo s pivotnim elementom. Promatrajući elemente slijeva nadesno, ukoliko je promatrani element manji ili jednak pivotnom elementu, tada povećavamo indeks *l*. Ako dođemo do elementa koji je veći od pivota, stajemo s povećavanjem indeksa *l*. Zatim promatramo elemente zdesna nalijevo. Ako je promatrani element veći od pivotnog elementa, tada smanjujemo indeks *r*, a ako je manji od pivota, tada stajemo sa smanjivanjem indeksa *r*. S obje strane smo pronašli elemente koji nisu u ispravnom poretku (lijevo manji od pivota, desno veći) te ih zamjenjujemo da bi došli na pravu stranu.

Ponavljajući ovaj postupak, sve ćemo elemente staviti na ispravnu stranu, tako da se konačno svi elementi manji ili jednaki od pivota nalaze lijevo od njega, a svi elementi veći od pivota se nalaze desno od njega. Uočimo da pivot tako dolazi na konačno mjesto u sortiranom nizu te funkcija *partition* vraća njegov indeks.

Pseudokod funkcije *partition* je:

```
partition(i, j, pivotindex) {
  pivot=A[pivotindex]
  l=i
  r=j
  while (l<r) {
    while (A[l]<=pivot) l++
    while (A[r]>pivot) r--
    if (l<r) then
      swap(A[l], A[r])
  }
  swap(A[pivotindex], A[r])
  return r
}
```

Slika 2.4 pokazuje korake rada upravo opisane funkcije *partition*. Pivotni element je obojan crvenom bojom, a elementi koje zamjenjujemo plavom bojom. Uočimo da se pivot na početku rada nalazi na indeksu 0, a na kraju rada na indeksu 4.



Slika 2.4: Primjer rada funkcije *partition*

Algoritam *quicksort* prima kao ulazne parametre indekse i i j, koji označavaju koji dio niza A ovim pozivom želimo sortirati. Na početku algoritma zovemo funkciju $findpivot$ te, ukoliko je pivot određen, pozivamo funkciju *partition*, koja će rasporediti elemente oko pivota. Zatim pozivamo algoritam *quicksort* na podnizu lijevo od pivota i na podnizu desno od pivota. Rezultat je uzlazno sortiran niz A.

Prikažimo u nastavku pseudokod *quicksort* algoritma:

```
quicksort(i,j) {
  pivotindex = findpivot(i,j)
  if pivotindex != 0 then
    k = partition(i, j, pivotindex)
    quicksort(i, k-1)
    quicksort(k, j)
}
```

Vremenska složenost quicksort algoritma

Neka je s $T_Q(n)$ označeno prosječno vrijeme izvršavanja *quicksort* algoritma za sortiranje n elemenata. Ako je $n = 1$ tada je $T_Q(1) = c$, za neku konstantu $c > 0$. Pretpostavimo da je $n > 1$ i da su vrijednosti svih elemenata različite. Tada će za funkcije $findpivot$ i *partition* trebati dn vremena, gdje je $d > 0$ konstanta. Nakon funkcija $findpivot$ i *partition* algoritam *quicksort* poziva samog sebe na 2 manja podniza. Dakle, dobivamo sljedeću gornju granicu za $T_Q(n)$:

$$T_Q(n) \leq T_R(n) + dn,$$

gdje je $T_R(n)$ prosječno vrijeme trajanja rekurzivnih poziva.

Odredimo sada $T_R(n)$. Neka lijevi podniz ima i od n elemenata, a desni $n-i$ elemenata. Vrijedi:

$$T_R(n) = \sum_{i=1}^{n-1} p_i \left[T_Q(i) + T_Q(n-i)\right],$$

gdje je p_i vjerojatnost da lijevi podniz ima točno i elemenata. S obzirom na pretpostavku da su vrijednosti svih elemenata različite, pivot se u lijevom podnizu mora nalaziti na indeksu 0 ili 1. Stoga je vjerojatnost p_i jednaka $\frac{2i}{n(n-1)}$. Time dobivamo:

$$T_Q(n) \leq \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} \left[T_Q(i) + T_Q(n-i)\right] + dn.$$

Iz jednakosti $\sum_{i=1}^{n-1} f(i) = \sum_{i=1}^{n-1} f(n-i)$ dobivamo relaciju:

$$\sum_{i=1}^{n-1} f(i) = \frac{1}{2} \sum_{i=1}^{n-1} \left[f(i) + f(n-i)\right]. \tag{2.6}$$

Neka je $f(i) = \frac{2i}{n(n-1)} \left[T(i) + T(n-i)\right]$. Tada, koristeći relaciju (2.6), dobivamo sljedeću nejednakost za $T_Q(n)$:

$$T_Q(n) \leq \frac{1}{2} \sum_{i=1}^{n-1} \left[\frac{2i}{n(n-1)} \left[T_Q(i) + T_Q(n-i)\right] + \frac{2(n-i)}{n(n-1)} \left[T_Q(n-i) + T_Q(i)\right] \right] + dn$$

$$\leq \frac{1}{n-1}\sum_{i=1}^{n-1}[T_Q(i)+T_Q(n-i)]+dn \leq \frac{2}{n-1}\sum_{i=1}^{n-1}T_Q(i)+dn.$$

Sada dokazujem matematičkom indukcijom da postoji konstanta $a > 0$, takva da vrijedi:

$$T_Q(n) \leq an\log n,\ \forall n \geq 2.$$

1. Baza indukcije: Neka je $n = 2$. Tada imamo:

   $$T_Q(2) \leq 2T_Q(1)+2d \leq 2c+2d.$$

   Iz $T_Q(2) \leq a2\log 2$ slijedi:

   $$c+d \leq a\log 2,$$

   odnosno,

   $$a \geq \frac{c+d}{\log 2}.$$

2. Neka je $n \in \mathbb{N}$, $n > 2$. Pretpostavimo da za svaki $i \in \mathbb{N}$, takav da je $i < n$, vrijedi $T(i) \leq ai\log i$. Dokažimo da tada vrijedi $T(n) \leq an\log n$.

3. Korak indukcije:

   $$T_Q(n) \leq \frac{2}{n-1}\sum_{i=1}^{n-1}T_Q(i)+dn \leq \frac{2a}{n-1}\sum_{i=1}^{n-1}i\log i+dn. \tag{2.7}$$

   Sumu $\sum_{i=1}^{n-1} i\log i$ ocjenjujemo odozgo:

   $$\sum_{i=1}^{n-1}i\log i \leq \sum_{i=1}^{n/2}i\log i+\sum_{i=n/2+1}^{n-1}i\log i \leq \sum_{i=1}^{n/2}i(\log n-1)+\sum_{i=n/2+1}^{n-1}i\log n$$

   $$\leq \frac{n}{4}\left(\frac{n}{2}+1\right)\log n-\frac{n}{4}\left(\frac{n}{2}+1\right)+\frac{3}{4}n\left(\frac{n}{2}-1\right)\log n \leq \left(\frac{n^2}{2}-\frac{n}{2}\right)\log n-\left(\frac{n^2}{8}+\frac{n}{4}\right).$$

   Ovu gornju ogradu uvrstimo u (2.7) i dobivamo:

   $$T_Q(n) \leq an\log n-\frac{an}{4}-\frac{an}{2(n-1)}+dn.$$

   Ako je $a \geq 4c$ tada je $-\frac{an}{4}-\frac{an}{2(n-1)}+dn \leq 0$, pa vrijedi:

   $$T_Q(n) \leq an\log n.$$

   Dakle, konačno dobivamo $T_Q(n) \in O(n\log n)$.

2.4 Množenje velikih cijelih brojeva

Promotrimo sada malo drugačiju primjenu metode "podijeli pa vladaj" za množenje velikih cijelih brojeva. Općenito, vremenska složenost računanja produkta dva cijela broja prikaziva u računalu je $O(1)$. Međutim, prilikom množenja cijelih brojeva proizvoljne duljine to više ne vrijedi, već se vremenska složenost povećava. Klasičan algoritam množenja dva velika cijela broja ima vremensku složenost $O(n^2)$, gdje je n broj znamenaka cijelog broja, no koristeći metodu podijeli pa vladaj, kao što će biti opisano u ovom poglavlju, ta složenost može biti značajno smanjena.

Slika 2.5 prikazuje jedan primjer množenja dva binarna broja koristeći klasičan algoritam. Neka su $x = 1001$ i $y = 1011$ dva binarna broja. Njihov produkt računamo tako da svaku znamenku iz y pomnožimo s brojem x te smo tako izračunali parcijalne produkte. Zbrojimo sve parcijalne produkte i dobijemo produkt xy.

```
  1001 x 1011
 ------------
    1001
     0000
      1001
+      1001
 ------------
   1100011
```

Slika 2.5: Primjer množenja klasičnim algoritmom

Pokušajmo sada poboljšati algoritam tako da n-bitne brojeve razdvojimo na dva dijela duljine $\frac{n}{2}$ bitova. Radi jednostavnosti, pretpostavimo da je n potencija broja 2. Neka su X i Y dva n-bitna cijela broja. Cijele brojeve X i Y možemo raspisati na sljedeći način:

$$X = \boxed{\text{A}}\boxed{\text{B}} = A2^{\frac{n}{2}} + B$$

$$Y = \boxed{\text{C}}\boxed{\text{D}} = C2^{\frac{n}{2}} + D.$$

Na primjer, cijeli broj $Z = 10110001$ možemo podijeliti na dva 4-bitna broja A i B, gdje je $A = 1011$ i $B = 0001$.

Produkt brojeva X i Y jednak je:

$$XY = AC2^n + (AD + BC)\,2^{\frac{n}{2}} + BD. \tag{2.8}$$

Dakle, koristeći jednadžbu (2.8), za računanje produkta potrebno je izračunati 4 produkata $\frac{n}{2}$-bitnih brojeva (to su: AC, AD, BC i BD), 3 zbrajanja s najviše $2n$ bitova i 2

množenja s 2^n i $2^{\frac{n}{2}}$. Pritom, množenje brojeva s 2^n je, zapravo, jednostavno pomicanje n bitova ulijevo.

Vremenska složenost

Uočimo da zbrajanja i pomicanja bitova zahtijevaju $O(n)$ koraka, a množenja $\frac{n}{2}$-bitnih brojeva se računaju prema gore opisanom postupku. Time dobivamo sljedeću rekurzivnu relaciju za računanje vremenske složenosti:

$$T(n) = \begin{cases} d, & ako\ je\ n = 1 \\ 4T(\frac{n}{2}) + bn, & ako\ je\ n > 1, \end{cases}$$

za neku konstantu b i za $d > 0$.

Rješavanjem ove rekurzivne relacije dobivamo da je složenost opisanog algoritma jednaka $O(n^2)$. Budući da je složenost ista kao kod klasičnog algoritma, mogli bismo reći da smo bespotrebno zakomplicirali račun, a dobili istu složenost. Odnosno, mogli bismo reći kako nismo ništa postigli. Međutim, pokušajmo poboljšati algoritam tako da trebamo samo 3 rekurzivna poziva.

Poboljšanje algoritma

Razmotrimo računanje broja $AD + BC$ koristeći sljedeći identitet:

$$AD + BC = (A + B)(C + D) - AC - BD.$$

Time imamo samo 3 množenja. Tako, produkt brojeva X i Y, smanjimo na 3 množenja cijelih brojeva veličine $\frac{n}{2}$ i 6 zbrajanja i oduzimanja, te dobivamo sljedeću rekurziju:

$$T(n) = \begin{cases} d, & ako\ je\ n = 1 \\ 3T(\frac{n}{2}) + bn, & ako\ je\ n > 1, \end{cases}$$

za neku konstantu b i za $d > 0$.

Riješimo sada ovu rekurziju kao što je opisano u prethodnom poglavlju. Prvo uvodimo supstituciju $n = 2^k$, odnosno, $k = \log_2(n)$. Time rekurzija ima oblik:

$$T(2^k) = 3T(2^{k-1}) + b2^k,$$

odnosno, uz oznaku $T(n) = T(2^k) = t_k$:

$$t_k = 3t_{k-1} + b2^k. \tag{2.9}$$

Tada je karakteristična jednadžba:

$$(x-3)(x-2)=0,$$

te je opće rješenje rekurzije jednako:

$$t_k = c_1 2^k + c_2 3^k, \tag{2.10}$$

ili

$$T(n) = c_1 n + c_2 3^{\log_2 n}.$$

Odredimo konstante c_1 i c_2. Konstantu c_1 odredimo tako da rješenje rekurzije (2.10) uvrstimo u (2.9). Dobivamo:

$$c_1 2^k + c_2 3^k = 3(c_1 2^{k-1} + c_2 3^{k-1}) + b2^k,$$

odnosno,

$$c_1 2^k = 3c_1 2^{k-1} + b2^k.$$

Sređivanjem ove jednadžbe konačno dobivamo $c_1 = -2b$. Iz početnog uvjeta $T(1) = d$ dobijemo:

$$c_2 = d + 2b.$$

Uz uvjete $b, d \geq 0$ očito vrijedi $c_2 \geq 0$, te budući da vrijedi $3^{\log_2 n} = n^{\log_2 3}$, dobivamo:

$$T(n) \in O(n^{\log_2 3}).$$

Dakle, složenost je $O(n^{\log_2 3}) = O(n^{1.59})$, što nam daje izuzetan napredak u odnosu na klasičnu metodu.

Na kraju pogledajmo pseudokod navedenog algoritma:

```
Multiplication(X, Y, n) {
  s:=sign(X) * sign(Y);
  X:=abs(X);
  Y:=abs(Y);
  if n=1 then
    if X=1 and Y=1 then
      return s
    else
      return 0
  else {
      A:=lijevih n/2 bitova od X;
      B:=desnih n/2 bitova od X;
      C:=lijevih n/2 bitova od Y;
```

```
        D:=desnih n/2 bitova od Y;
        m1:= Multiplication(A,C,n/2);
        m2:= Multiplication(A–B,D–C,n/2);
        m3:= Multiplication(B,D,n/2);
        return (s * (m1*2^n + (m1+m2+m3) * 2^(n/2) + m3))
    }
}
```

2.5 Množenje matrica

Kao zadnji primjer opisujem problem množenja matrica. Opisat ću dvije verzije algoritma oblikovanog metodom "podijeli pa vladaj" koji rješava problem množenja kvadratnih matrica. Prva verzija će imati istu složenost kao klasičan algoritam množenja matrica, no druga verzija, poznata kao Strassenov algoritam, će prikazivati efikasniji algoritam.

Neka su A i B dvije kvadratne matrice reda $n \in \mathbb{N}$. Cilj nam je izračunati njihov produkt $C = AB$.

Klasični algoritam

Koristeći klasičnu metodu, produkt matrica C računamo koristeći forumulu:

$$C(i, j) = \sum_{k=1}^{n} A(i, k)B(k, j).$$

Ova formula zahtijeva n^3 množenja i $n^3 - n^2$ zbrajanja. Stoga je vremenska složenost ovog algoritma $O(n^3)$.

Prva verzija "podijeli pa vladaj" algoritma

Radi jednostavnosti, pretpostavimo da je n potencija broja 2. Tada svaku od matrica A, B i C možemo podijeliti na 4 matrice reda $\frac{n}{2}$, ovako:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \qquad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

Tada produkt C računamo na ovaj način:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

Ovakvo računanje produkta $C = AB$ zahtijeva 8 množenja i 4 zbrajanja matrica reda $\frac{n}{2}$. Označimo li s $T(n)$ broj množenja i zbrajanja potrebnih za računanje produkta matrica reda n, dobivamo da je vremenska složenost ovog algoritma opisana rekurzivnom relacijom:

$$T(n) = 8T\left(\frac{n}{2}\right) + 4\left(\frac{n}{2}\right)^2.$$

Rješavanjem ove relacije dobivamo vremensku složenost $O(n^3)$. Zaključujemo da ovakvim načinom oblikovanja algoritma ne dobivamo efikasniji algoritam u usporedbi s klasičnom metodom.

Strassenov algoritam

Konstruirajmo sada drugu verziju algoritma oblikovanog metodom "podijeli pa vladaj". Neka su matrice A, B i C reda n podijeljene kao što je već opisano. Cilj je oblikovati algoritam koji koristi manje množenja matrica reda $\frac{n}{2}$, u odnosu na prvu verziju i tako dobiti efikasniji algoritam.

Prije računanja samog produkta C, potrebno je izračunati sljedeće produkte:

$$\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22})B_{11} \\
P_3 &= A_{11}(B_{12} - B_{22}) \\
P_4 &= A_{22}(B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{12})B_{22} \\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}).
\end{aligned}$$

Produkt C računamo ovako:

$$C = \begin{bmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 + P_3 - P_2 + P_6 \end{bmatrix}.$$

U nastavku je prikazan pseudokod Strassenovog algoritma:

```
Strassen(A,B)
  if n=1 then return (A · B)
  else
    compute A11, B11, ..., A22, B22
    P1 = Strassen(A11 + A22, B11 + B22)
    P2 = Strassen(A21 + A22, B11)
    P3 = Strassen(A11, B12 − B22)
    P4 = Strassen(A22, B21 − B11)
    P5 = Strassen(A11 + A12, B22)
    P6 = Strassen(A21 − A11, B11 + B12)
    P7 = Strassen(A12 − A22, B21 + B22)
    C11 = P1 + P4 − P5 + P7
    C12 = P3 + P5
    C21 = P2 + P4
    C22 = P1 + P3 − P2 + P6
    return C
```

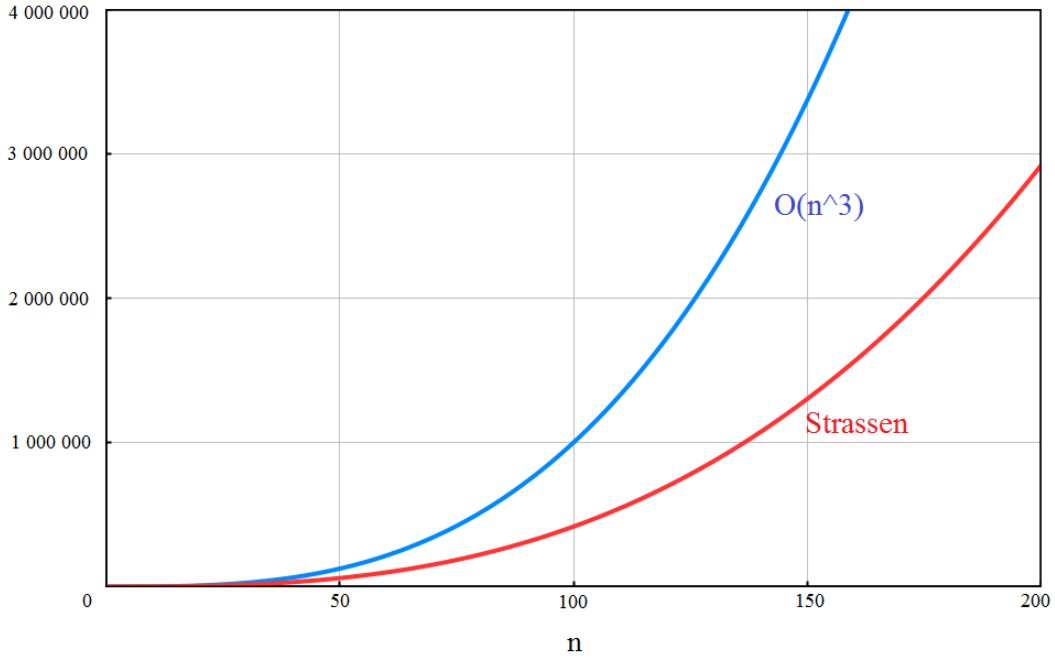
Vremenska složenost Strassenovog algoritma

Kako bismo smanjili vremensku složenost ovog algoritma, cilj nam je bio smanjiti broj množenja matrica i time povećati broj zbrajanja i oduzimanja. Ovaj algoritam koristi 7 množenja i 18 zbrajanja matrica reda $\frac{n}{2}$. Dakle, rekurzivna relacija koja opisuje vremensku složenost ovog algoritma je:

$$T(n) = \begin{cases} m, & ako\ n = 1 \\ 7T(\frac{n}{2}) + 18(\frac{n}{2})^2 a, & ako\ n > 1. \end{cases}$$

Time dobivamo vremensku složenost jednaku $O(n^{\log 7}) = O(n^{2.81})$.

Slika 2.6 prikazuje efikasnost Strassenovog algoritma u odnosu na klasičan algoritam i prethodno opisan algoritam oblikovan metodom "podijeli pa vladaj", čije su vremenske složenosti $O(n^3)$.



Slika 2.6: Usporedba Strassenovog i klasičnog algoritma

Poglavlje 3

Testiranje

Testiranje je pokusno izvođenje programa, pisanog u određenom programskom jeziku, sa svrhom verifikacije i validacije. Verifikacija je provjera radi li program prema određenim specifikacijama, a validacija je provjera odgovara li program potrebama korisnika. Dakle, testiranjem se provjerava radi li program ispravno te se tako mogu otkriti i otkloniti moguće greške.

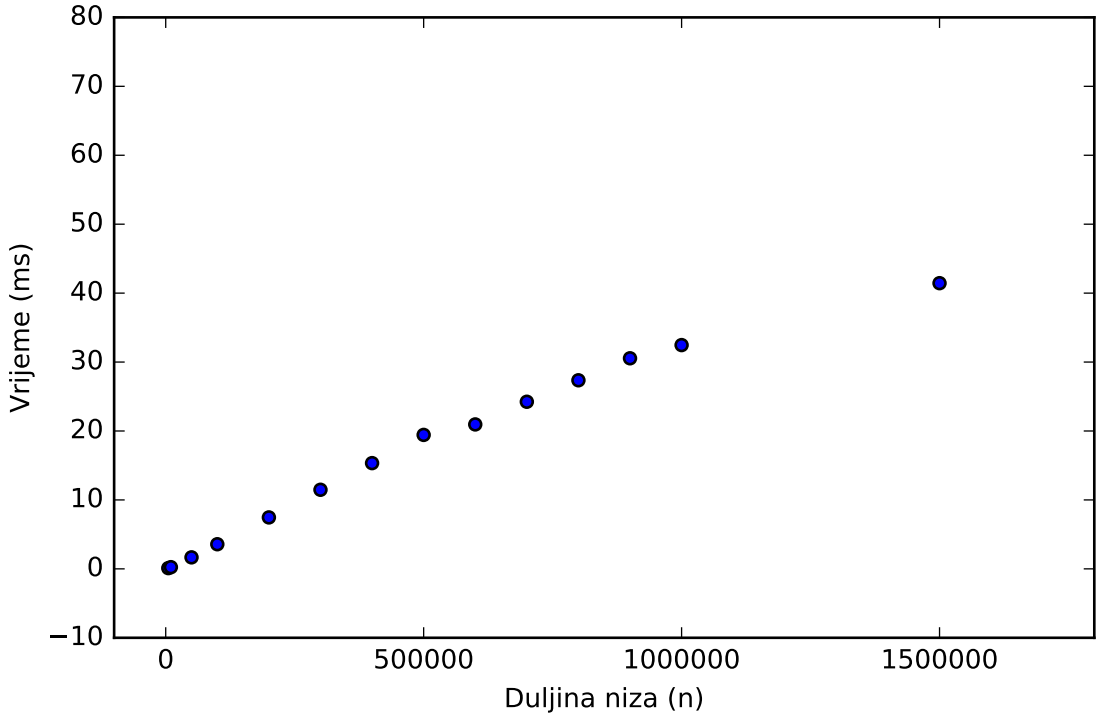
U ovom diplomskom radu ću testirati programe tako što ću mjeriti vrijeme izvršavanja programa, ovisno o veličini ulaznih podataka. Programi za testiranje su pisani u programskom jeziku C++. Testiranje je izvršeno na računalu s 4 GB radne memorije i dvojezgrenim procesorom čija je brzina 1.9 GHz.

3.1 Testiranje algoritma binarnog pretraživanja

Za početak, testirajmo algoritam binarnog pretraživanja opisan u poglavlju 2.1. Prisjetimo se, binarno pretraživanje provjerava nalazi li se cijeli broj x u nizu uzlazno sortiranih cijelih brojeva A.

Ulazni podaci, odnosno, niz cijelih brojeva A i cijeli broj x su slučajno generirani cijeli brojevi iz intervala [0, 100000]. Algoritam sam testirala na nizovima duljine 5 000, 10 000, 50 000, 100 000, 200 000, 300 000, 400 000, 500 000, 600 000, 700 000, 800 000, 900 000, 1 000 000, 1 500 000, 2 000 000, 2 500 000 i 3 000 000. Za svaku od navedenih duljina niza mjerila sam vrijeme izvršavanja 100 puta, te odredila prosječno vrijeme izvršavanja. Rezultati testiranja su prikazani u tablici 3.1 i točkastom grafu, koji se nalazi na slici 3.1.

Duljina ulaznog niza cijelih brojeva (n)	**Prosječno vrijeme izvršavanja (ms)**
5 000	0.10743
10 000	0.24635
50 000	1.66885
100 000	3.57241
200 000	7.4624
300 000	11.4729
400 000	15.339
500 000	19.4324
600 000	20.9522
700 000	24.2412
800 000	27.3512
900 000	30.5474
1 000 000	32.4619
1 500 000	41.444



Slika 3.1: Graf testiranja algoritma binarnog pretraživanja

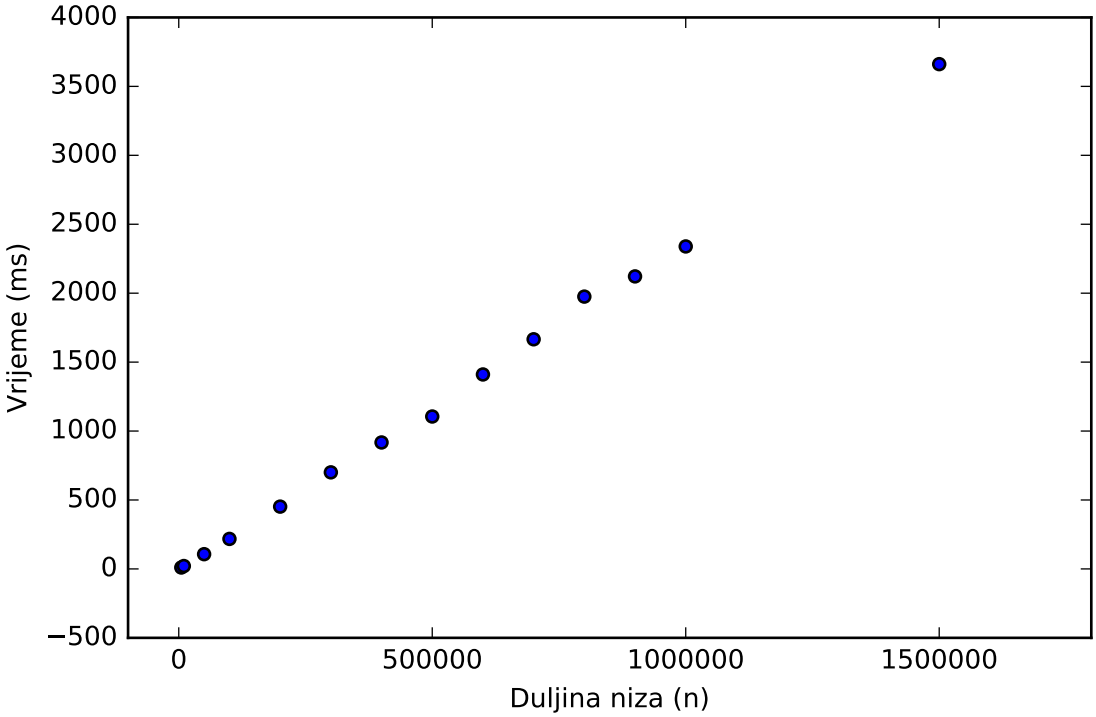
3.2 Testiranje mergesort algoritma

Prilikom testiranja mergesort algoritma za sortiranje nizova cijelih brojeva koristila sam, također, program algoritma pisan u programskom jeziku C++. Testiranje sam izvršavala na nizovima duljine 5 000, 10 000, 50 000, 100 000, 200 000, 300 000, 400 000, 500 000, 600 000, 700 000, 800 000, 900 000, 1 000 000 i 1 500 000, čije su vrijednosti slučajno generirani cijeli brojevi iz intervala $[0, 100000]$.

Postupak mjerenja vremena izvršavanja mergesort algoritma sam ponavljala 100 puta za svaku duljinu ulaznog niza, te izračunala prosječno vrijeme izvršavanja algoritma. Tablica 3.2 prikazuje prosječno vrijeme izvršavanja programa u milisekundama.

Duljina ulaznog niza cijelih brojeva (n)	**Prosječno vrijeme izvršavanja (ms)**
5 000	10.2275
10 000	20.9238
50 000	106.916
100 000	217.668
200 000	451.86
300 000	700.432
400 000	917.22
500 000	1 105.49
600 000	1 210.45
700 000	1 655.61
800 000	1 975.18
900 000	2 121.71
1 000 000	2 339.02
1 500 000	3 661.33

Slika 3.2 prikazuje točkasti graf testiranja mergesort algoritma na kojem je vidljiv odnos duljine niza i vremena izvršavanja algoritma.

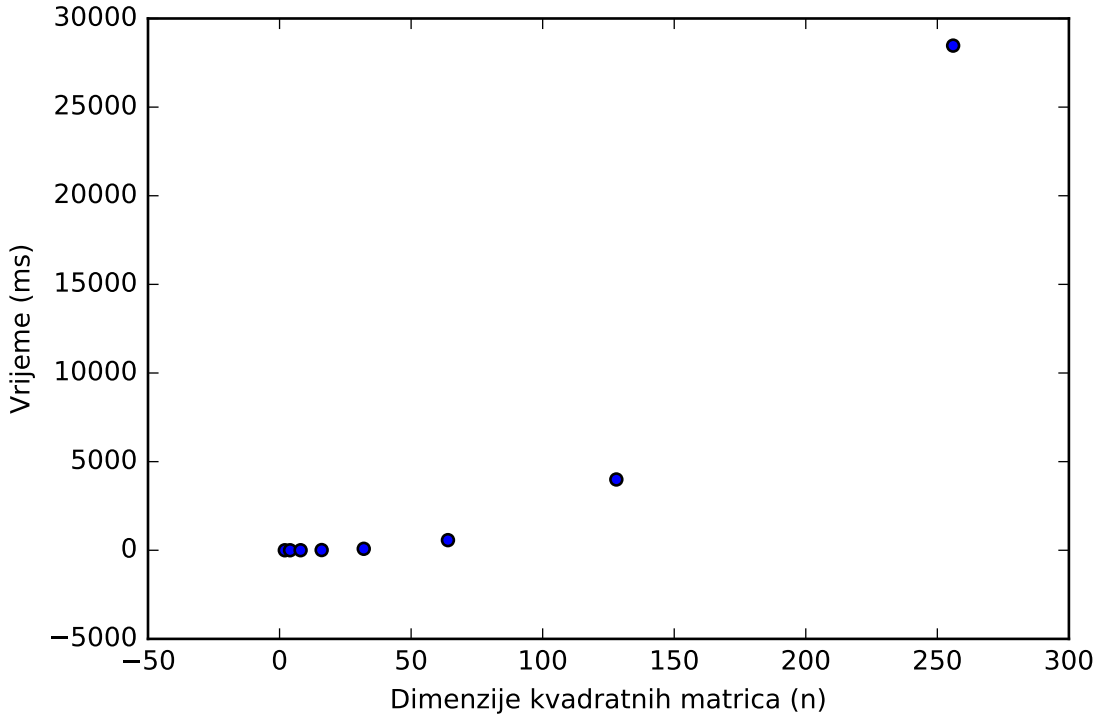


Slika 3.2: Graf testiranja mergesort algoritma

3.3 Testiranje Strassenovog algoritma

Kao što je navedeno u prethodnom poglavlju, Strassenov algoritam koristimo za množenje kvadratnih matrica reda $n \in \mathbb{N}$, pri čemu je n potencija broja 2. Testiranje sam izvršavala na matricama reda 2, 4, 8, 16, 32, 64, 128 i 256. Analogno kao kod mergesort algoritma i algoritma binarnog pretraživanja, postupak mjerenja vremena izvršavanja ponavljala sam 100 puta za svaku vrijednost reda matrice, te izračunala prosječno vrijeme izvršavanja. Rezultati mjerenja navedeni su u tablici 3.3 i prikazani na točkastom grafu. Vrijednosti kvadratnih matrica su slučajno generirani brojevi iz intervala $[1, 10]$.

Red kvadratnih matrica (n)	**Prosječno vrijeme izvršavanja (ms)**
2	0.04854
4	0.39966
8	1.97884
16	11.6425
32	82.4216
64	570.436
128	3 992.88
256	28 469.8



Slika 3.3: Graf testiranja Strassenovog algoritma

Poglavlje 4

Prilozi

4.1 BinarySearch.cpp

```
#ifndef BINARYSEARCH_C_INCLUDED
#define BINARYSEARCH_C_INCLUDED

#include <iostream>      // std::cout
#include <algorithm>     // std::binary_search, std::sort
#include <vector>        // std::vector
#include <ctime>

bool myfunction (int i,int j) { return (i<j); }

int BinarySearch(std::vector<int> A, int start,
                                 int end, int x) {
    if(start>end) return -1;
    else {
        int mid=(start+end)/2;
        if(x==A[mid]) return mid;
        else {
            if(x<A[mid])
                return BinarySearch(A, start, mid-1, x);
            else
                return BinarySearch(A, mid+1, end, x);
        }
    }
}
```

```
int main () {
    int broj, brojElemenata, x, i=0;
    std::vector<int> A;
    std::vector<double> vrijeme;
    std::cout << "Unesi broj elemenata: ";
    std::cin >> brojElemenata;

    while(i<100) {
        clock_t p;
            // slucajno odaberemo broj koji cemo traziti
        x = (int) rand() % 100000;
            // slucajno generiramo niz cijelih brojeva
        for(int j=0;j<brojElemenata;j++) {
            broj = (int) rand() % 100000;
            A.push_back(broj);
        }
            // sortiramo niz cijelih brojeva
        std::sort(A.begin(), A.end());

        p = clock();
        BinarySearch(A, 0, brojElemenata-1, x);
        p = clock() - p;
            // izracunamo vrijeme izvrsavanja u sekundama
        double sekunde = ((double)p)/CLOCKS_PER_SEC;
        vrijeme.push_back(sekunde*1000);
        A.clear();
        i++;
    }
    double sumaMilisekundi=0;
    for(int i=0;i<100;i++) sumaMilisekundi +=vrijeme[i];
    std::cout << "Prosjecno vrijeme izvrsavanja = "
        << (sumaMilisekundi/100) << " ms"<< std::endl;
    return 0;
}

#endif // BINARYSEARCH_C_INCLUDED
```

4.2 Mergesort.cpp

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <ctime>
#include <cstdlib>

std::vector<int> A;       // ulazni niz cijelih brojeva

void merge(int i, int m, int j) {
    std::vector<int> C;
    int p=i, q=m+1, r=i;
    while(p<=m && q<=j) {
        if(A[p]<=A[q]) {
            C.push_back(A[p]);
            p++;
        }
        else {
            C.push_back(A[q]);
            q++;
        }
    }
    while(p<=m) {
        C.push_back(A[p]);
        p++;
    }
    while(q<=j) {
        C.push_back(A[q]);
        q++;
    }
    for(int k=0;k<C.size();k++) {
        A[r+k] = C[k];
    }
}

void mergesort(int i, int j) {
    if(i==j) return;
    int m = (i+j)/2;
```

```
    mergesort(i, m);
    mergesort(m+1, j);
    merge(i, m, j);
}

int main () {
    int broj, brojElemenata, x, i=0;
    clock_t pocetak, kraj;
    std::vector<double> vrijeme;
    std::cout << "Unesi broj elemenata niza: ";
    std::cin >> brojElemenata;

    while(i<100) {
        // slucajno generiraj niz brojeva
        for(int j=0;j<brojElemenata;j++) {
            broj = (int) rand() % 100000;
            A.push_back(broj);
        }

        pocetak = clock();
        mergesort(0, brojElemenata-1);
        kraj = clock() - pocetak;

        double sekunde = ((double)(kraj))/CLOCKS_PER_SEC;
        vrijeme.push_back(sekunde*1000);
        A.clear();
        i++;
    }

    double sumaMilisekundi=0;
    for(int i=0;i<100;i++)
        sumaMilisekundi +=vrijeme[i];
    std::cout << "Prosjecno vrijeme izvrsavanja = "
        << (sumaMilisekundi/100) << " ms." << std::endl;
    return 0;
}
```

4.3 Strassen.cpp

```
#include <iostream>
#include <vector>
#include <ctime>
#include <array>

class Matrix {
public:
    int n; // red matrice
    std::vector<int> Data; // vrijednosti matrice
    Matrix() = default;
    Matrix(int N) {
        n = N;
    }
    int operator()(int i, int j) {
        return Data[i*n+j];
    }
};

void ispisMatrice(Matrix A) {
    int i, j;
    for(int i=0;i<A.n;i++) {
        for(int j=0;j<A.n;j++) {
            std::cout << A(i,j) << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

Matrix operator+(Matrix& A, Matrix& B) {
        Matrix C;
        C.n = A.n;
        for(int i=0;i<A.n;i++) {
            for(int j=0;j<A.n;j++) {
                C.Data.push_back(A(i,j)+B(i,j));
            }
        }
```

```
        return C;
    }

Matrix operator-(Matrix& A, Matrix& B) {
    Matrix C;
    C.n = A.n;
    for(int i=0;i<A.n;i++) {
        for(int j=0;j<A.n;j++) {
            C.Data.push_back(A(i,j)-B(i,j));
        }
    }
    return C;
}
    // A i B su kvadratne matrice istog reda
Matrix Strassen(Matrix A, Matrix B) {
    Matrix C;
    C = A.n;
    if(A.n==1) {
        int x = A(0,0)*B(0,0);
        C.Data.push_back(x);
        return C;
    }
    else {
        Matrix A11, A12, A21, A22;
        A11.n = A12.n = A21.n = A22.n = A.n/2;
        for(int i=0;i<A.n/2;i++) {
            for(int j=0;j<A.n/2;j++) {
                A11.Data.push_back(A(i,j));
            }
        }
        for(int i=0;i<A.n/2;i++) {
            for(int j=A.n/2;j<A.n;j++) {
                A12.Data.push_back(A(i,j));
            }
        }
        for(int i=A.n/2;i<A.n;i++) {
            for(int j=0;j<A.n/2;j++) {
                A21.Data.push_back(A(i,j));
            }
```

```
}
for(int i=A.n/2;i<A.n;i++) {
    for(int j=A.n/2;j<A.n;j++) {
        A22.Data.push_back(A(i,j));
    }
}

Matrix B11, B12, B21, B22;
B11.n = B12.n = B21.n = B22.n = B.n/2;
for(int i=0;i<B.n/2;i++) {
    for(int j=0;j<B.n/2;j++) {
        B11.Data.push_back(B(i,j));
    }
}
for(int i=0;i<B.n/2;i++) {
    for(int j=B.n/2;j<B.n;j++) {
        B12.Data.push_back(B(i,j));
    }
}
for(int i=B.n/2;i<B.n;i++) {
    for(int j=0;j<B.n/2;j++) {
        B21.Data.push_back(B(i,j));
    }
}
for(int i=B.n/2;i<B.n;i++) {
    for(int j=B.n/2;j<B.n;j++) {
        B22.Data.push_back(B(i,j));
    }
}

Matrix P1, P2, P3, P4, P5, P6, P7;
P1.n = P2.n = P3.n = P4.n =
    P5.n = P6.n = P7.n = A.n/2;
Matrix a = A11 + A22, b= B11+B22;
P1 = Strassen(a,b);
P2 = Strassen(A21+A22, B11);
P3 = Strassen(A11, B12-B22);
P4 = Strassen(A22, B21-B11);
P5 = Strassen(A11+A12, B22);
```

```
        P6 = Strassen(A21-A11, B11+B12);
        P7 = Strassen(A12-A22, B21+B22);

        Matrix C11, C12, C21, C22;
        C11.n = C12.n = C21.n = C22.n = B.n/2;
        C.n = A.n;
        C11 = P1 + P4; // C11 = P1 + P4 - P5 + P7;
        C11 = C11 - P5;
        C11 = C11 + P7;
        C12 = P3 + P5;
        C21 = P2 + P4;
        C22 = P1 + P3; // C22 = P1 + P3 - P2 + P6;
        C22 = C22 - P2;
        C22 = C22 + P6;

        for(int i=0;i<C11.n;i++) {
            for(int j=0;j<C11.n;j++) {
                C.Data.push_back(C11(i,j));
            }
            for(int j=0;j<C11.n;j++) {
                C.Data.push_back(C12(i,j));
            }
        }
        for(int i=0;i<C21.n;i++) {
            for(int j=0;j<C21.n;j++) {
                C.Data.push_back(C21(i,j));
            }
            for(int j=0;j<C22.n;j++) {
                C.Data.push_back(C22(i,j));
            }
        }
        return C;
    }
}

int main(void) {
    int a, b, n, i=0;
    clock_t c;
    std::vector<double> vrijeme;
```

```
    std::cout << "Unesi dimenzije kvadratnih matrica: ";
    std::cin >> n;
    while(i<100) {
        Matrix A(n), B(n);
            // slucajno generiranje matrica A i B
        for(int i=0;i<n;i++) {
            for(int j=0;j<n;j++) {
                a = (int) rand() % 10;
                A.Data.push_back(a);
                b = (int) rand() % 10;
                B.Data.push_back(b);
            }
        }

        c = clock();
        Strassen(A,B);
        c = clock() - c;
            // izracunamo vrijeme izvrsavanja u sekundama
        double sekunde = ((double)(c))/CLOCKS_PER_SEC;
        vrijeme.push_back(sekunde);
        i++;
    }
    double sumaSekundi=0;
    for(int i=0;i<100;i++) sumaSekundi +=vrijeme[i];
    std::cout << "Prosjek izvrsavanja (u milisekundama) = "
            << (sumaSekundi*10) << std::endl;
    return 0;
}
```

Bibliografija

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullmann, *Data Structures and Algorithms*, Addison-Wesley, Reading, Massachusetts, 1983. (Reprinted with corrections, 1987.).

[2] J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson Education, Inc., Boston, Massachusetts, 2006.

[3] *Merge sort*, dostupno na `https://en.wikipedia.org/wiki/Merge_sort` (rujan 2016.)

[4] M. H. Alsuwaiyel, *Algorithms — Design Techniques and Analysis*, World Scientific, Singapore, 2003.

[5] *Module II: Programming abstractions and analysis*, dostupno na `https://puzzle.ics.hut.fi/ICS-A1120/2015/notes/module-abstractions.html#module2` (rujan 2016.)

[6] M. Vuković, *Složenost algoritama*, skripta (PMF - Matematički odjel Zagreb, 2015.).

[7] R. Johnsonbaugh, M. Schaefer, *Algorithms*, Pearson Education International, Upper Saddle River, New Jersey, 2004.

[8] S. Singer, *Uvod u složenost*, dostupno na `http://degiorgi.math.hr/oaa/scans/pog_1.pdf` (rujan 2016.)

Sažetak

Ovaj diplomski rad opisuje metodu ”podijeli pa vladaj” koja se koristi za oblikovanje algoritama.

Rad je podijeljen u tri poglavlja. Prvo poglavlje detaljnije opisuje spomenutu metodu, a navedeni su i koraci algoritama oblikovani ovom metodom. Također, u ovom poglavlju opisane su rekurzivne relacije, koje koristimo prilikom analize vremenske složenosti algoritama opisanih u drugom poglavlju.

Drugo poglavlje opisuje algoritam binarnog pretraživanja, mergesort algoritam, quicksort algoritam, algoritam množenja velikih cijelih brojeva te algoritam množenja matrica koji su oblikovani navedenom metodom. Za navedene algoritme je analizirana njihova vremenska složenost te je naveden njihov pseudokod.

U trećem poglavlju rada implementirani su neki od navedenih algoritama u programskom jeziku C++. Algoritmi su testirani mjerenjem vremena izvršavanja u odnosu na veličinu ulaznih podataka, a rezultati testiranja su prikazani tablicom i grafom.

Dakle, ovim diplomskom radom dolazimo do zaključka da je proučavana metoda ”podijeli pa vladaj” vrlo važna metoda dizajniranja algoritama koja se u praksi često koristi. Ovom metodom se mogu dobiti efikasni algoritmi za rješavanje nekih problema te je vrlo jednostavna i zanimljiva za korištenje.

Summary

This master thesis describes the ”divide and conquer” method which is used in the design of algorithms.

The thesis is divided into three chapters. The first chapter describes the basic prinicples of ”divide and conquer”, and general steps of the algorithms designed by this method. Also, in this chapter, there is a brief review of main results from the theory of recursive relations, that are used in the analysis of time complexity of the algorithms mentioned in the second chapter.

The second chapter contains several algorithms which are designed by the ”divide and conquer” method: the binary search algorithm, mergesort, quicksort, multiplication of large integers, and multiplication of matrices. For these algorithms, there is an analysis of their time complexity and their pseudocode.

In the third chapter of this thesis, we give implementations of some of the before mentioned algorithms in the C++ programming language. The algorithms are tested by measuring the execution time for various sizes of the input data. The results of each test are presented in table and graphic forms.

This master thesis on the ”divide and conquer” method concludes that this is a very important method for the design of algorithms. This method is widely used in practice because it can produce efficient algorithms for solving some of the problems. The method is also very simple and interesting.

Životopis

Moje ime je Petra Penzer. Rođena sam 21. studenog 1991. godine u Virovitici. Nakon završene osnovne škole, koju sam pohađala u Pitomači, upisala sam četverogodišnju farmaceutsku srednju školu na Zdravstvenom učilištu u Zagrebu, koju sam završila s odličnim uspjehom.

Godine 2010. upisala sam sveučilišni preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu, koji sam završila 15. rujna 2014. godine.

Diplomski studij matematike i računarstva na Prirodoslovno-matematičkom fakultetu sam upisala 2014. godine.