SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Sanja Budić

METODA *PODIJELI PA VLADAJ*

ZAVRŠNI RAD

Varaždin, 2010.

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Sanja Budić
Redoviti student
Broj indeksa: 35861/07-R
Smjer: Informacijski sustavi
Preddiplomski studij

METODA *PODIJELI PA VLADAJ*

ZAVRŠNI RAD

Mentor:
Tihomir Orehovački, mag. inf.

Varaždin, lipanj 2010.

Sadržaj

1. UVOD **1**

2. REKURZIJA **2**

2.1. ŠTO JE REKURZIJA? 2
2.2. REKURZIVNE FUNKCIJE 2
2.3. KARAKTERISTIKE REKURZIVNIH FUNKCIJA 3
2.4. KAKO RADE REKURZIVNI PROGRAMI? 4
2.5. USPOREDBA REKURZIVNIH I ITERATIVNIH PROGRAMA 6
2.6. VRSTE REKURZIVNIH FUNKCIJA 7

3. METODA *PODIJELI PA VLADAJ* **8**

3.1. KRATKA POVIJEST METODE *PODIJELI PA VLADAJ* 9
3.2. VRSTE METODE *PODIJELI PA VLADAJ* 9
3.3. IMPLEMENTACIJA 10
3.4. SLOŽENOST *PODIJELI PA VLADAJ* ALGORITAMA 10
3.4.1. Master teorem *11*
3.4.2. Akra-Bazzi metoda *12*

4. PREGLED NAJPOZNATIJIH PRIMJERA PRIMJENE METODE **13**

4.1. HANOJSKI TORNJEVI 13
4.1.1. Jednostavno rješenje *14*
4.1.2. Rekurzivno rješenje *15*
4.2. SORTIRANJE SPAJANJEM (MERGE SORT) 16
4.2.1. Složenost *19*
4.3. BRZO SORTIRANJE (QUICK SORT) 19
4.3.1. Složenost *22*
4.4. BINARNO PRETRAŽIVANJE (BINARY SEARCH) 22
4.4.1. Složenost *25*

5. PRIMJENA METODE U RJEŠAVANJU MODERNIH PROBLEMA **26**

6. ZAKLJUČAK **27**

7. LITERATURA **28**

8. POPIS SLIKA I TABLICA **30**

9. PRILOZI **31**

9.1. TORNJEVIHANOI.CPP 32
9.2. MERGESORT.CPP 33
9.3. QUICKSORT.CPP 35
9.4. BINARYSEARCH.CPP 36

1. Uvod

Metoda koja se često koristi u informatici kao i u ostalim srodnim granama, a naziva se *podijeli pa vladaj* (*engl.* divide and conquer) vrlo je zanimljiva i korisna. Ima mogućnost primjene u različitim područjima kao što su sortiranje, pretraživanje, razne transformacije, statističke analize i matematičke funkcije. Upravo ta raznolikost primjene i jednostavnost korištenja bila je glavna motivacija za nastajanje ovog rada.

U prvom dijelu govori se općenito o rekurziji, njezinim karakteristikama te primjeni i usporedbi rekurzivnih i nerekurzivnih (iterativnih) rješenja. Teorijski temelji opisani u ovom poglavlju dati će doprinos boljem i lakšem razumijevanju metode podijeli pa vladaj.

Sljedeće poglavlje donosi opis same metode uključujući njezinu povijest nastanka, obradu vrsta ove metode, implementaciju kao i određivanje asimptotske ocjene složenosti njezinih algoritama. Proučavanjem ovih aspekata moguće je bolje razumjeti osnovne koncepte metode podijeli pa vladaj.

U trećem se poglavlju daje uvid u jedne od najpoznatijih algoritama koji su primjer primjene ove metode. To su: hanojski tornjevi, sortiranje spajanjem (*engl*. Merge Sort), brzo sortiranje (*engl*. Quick Sort), binarno pretraživanje (*engl*. Binary Search). To ne znači da su ovo svi primjeri primjene ove metode. Postoje još mnogi, a ovo su samo neki od njih koji se smatraju važnijima u odnosu na ostale jer su jedni od prvih primjera primjene. Osim pregleda i objašnjenja tih algoritama, na kraju samog rada bit će priložene i njihove implementacije u programskom jeziku *C++*.

Poglavlje koje slijedi ukratko opisuje primjenu metode u nekim modernim problemima, konkretno opisuje primjenu metode kod rješavanja problema analize socijalnih mreža.

Ova se metoda prvenstveno koristi u informacijskim znanostima, no neke njezine odrednice iskoristive su i u svakodnevnom životu. Uvijek je lakše neki veliki problem, ukoliko je to moguće, podijeliti na nekoliko manjih problema istovrsne prirode koji se zatim lakše i jednostavnije rješavaju te se tako dolazi do konačnog rješenja glavnog problema, a to je upravo i temeljna svrha ove metode. Osim primjene u svakodnevnim situacijama važno je da se svatko tko se bavi analizom algoritama te izradom programskih rješenja za različite probleme upozna s osnovnim konceptima ove metode.

Svrha ovog rada je da uspješno, točno i razumljivo objasni spoznaje do kojih se došlo i time pomogne svima koje ova tema zanima te im je potrebna u rješavanju raznovrsnih problema.

2. Rekurzija

Budući da se rad metode *podijeli pa vladaj* zasniva na osnovnim pravilima rekurzije, potrebno je ponajprije shvatiti što je rekurzija te kako ona radi.

2.1. Što je rekurzija?

U programiranju i matematici često se koriste funkcije koje su rekurzivno definirane. Kad je funkcija rekurzivno definirana, to znači da ta funkcija poziva sama sebe. Postoje dvije vrste rekurzije [Metcalf, Reid, Cohen, 2004., str. 88-91]:

- direktna rekurzija,
- indirektna rekurzija.

Direktna rekurzija nastaje kad se u definiciji funkcije poziva ta ista funkcija. S druge strane, indirektna rekurzija nastaje kad jedna funkcija poziva neku drugu funkciju, a ova opet poziva funkciju iz koje je pozvana.

Može se zaključiti da je rekurzija u programiranju koncept koji omogućuje iterativno ponavljanje određenih akcija unutar funkcije ili procedure na način da određeni dio koda inicira ponovno pozivanje te iste funkcije ili procedure.

2.2. Rekurzivne funkcije

U prethodnom potpoglavlju definiran je pojam rekurzije te su pojašnjene njezine dvije osnovne vrste. U nastavku slijedi opis rekurzivnih funkcija potkrijepljen konkretnim primjerima primjene.

Definicija rekurzivne funkcije u pravilu se sastoji od dva dijela [Lee, 2008., str. 125]:

- temeljnog slučaja,
- pravila rekurzije.

Kao jednostavan primjer može se rekurzivno definirati funkcija koja opisuje faktorijelu.

Izraz je:

```
n! = n * (n-1) * (n-2) * .... * 3 * 2 * 1
```

Ako se bolje promotri gornji izraz, vidljivo je da se članovi mogu grupirati na sljedeći način:

```
n! = n * [(n-1) * (n-2) * .... * 3 * 2 * 1]
```

Time se zapravo dobije potpuno novi izraz:

```
n! = n * (n-1)!
```

Ako se sada promotri taj izraz, mogu se uočiti dva spomenuta dijela – *temeljni dio* (`n!=1 za n=0,1`) i *rekurzivni dio* (`n! = n * (n-1)! za n>1`).

Rekurzivne funkcije karakterizira jednostavan i efikasan zapis, s njima se puno lakše mogu riješiti problemi koji su već prirodno rekurzivni, a koje je kompliciranije riješiti nerekurzivnim putem.

2.3. Karakteristike rekurzivnih funkcija

Osnovno pitanje koje se postavlja kod rekurzija jest – može li ona pozivajući samu sebe ići u nedogled? Odgovor na to pitanje vrlo je jednostavan: *Naravno da ne, jer se tada proces izvršavanja rekurzije ne bi nikada završio*. Iz navedenog logički slijedi zaključak da kod takvih funkcija mora postojati **uvjet prekida** kad program zadanu operaciju bude izvršavao bez pozivanja samog sebe.

Ukoliko se pokušaju izračunati vrijednosti gore navedene funkcije za nekoliko brojeva, dobije se:

```
3! = 3 * 2 * 1 = 3 * 2!
2! = 2 * 1!
1! = 1
```

Iz ovoga proizlazi da rekurzivni pozivi prestaju onda kad argument funkcije postane jednak *1*, točnije, budući da je i *0!* jednako *1*, može se reći da rekurzivni pozivi prestaju za *0* ili *1*.

Jedan vrlo slikovit primjer koji pokazuje važnost postojanja uvjeta prekida rekurzije iz stvarnog života jest onaj s računima kreditnih kartica. Naime, kad se dobije uplatnica za dug koji je nastao korištenjem kreditne kartice, zašto se taj dug ne bi mogao platiti tom istom kreditnom karticom? Ovdje se jasno može vidjeti važnost postojanja uvjeta prekida rekurzije, jer isto kako bi se u stvarnom životu plaćali dugovi sa kreditne kartice kreditnom karticom, tako bi i rekurzija sama sebe pozivala do beskonačnosti.

Jedan od glavnih nedostataka rekurzivnih funkcija jest to što takve funkcije zauzimaju više memorijskog prostora, a uglavnom i zahtijevaju više vremena za izvođenje od odgovarajuće iterativne funkcije.

Razne se matematičke operacije mogu implementirati primjenom rekurzije. Međutim, kod primjene rekurzija treba biti krajnje oprezan. Budući da se lokalne varijable unutar funkcija i adrese s kojih se obavlja poziv funkcije pohranjuju na stog, preveliki broj uzastopnih poziva funkcije unutar te iste funkcije može vrlo brzo popuniti stog i onemogućiti daljnje izvođenje programa. Mnogo je sigurnije, a često i efikasnije, takve operacije implementirati jednostavnim petljama.

Rekurzije mogu biti vrlo efikasno sredstvo za rukovanje posebnim strukturama podataka, kao što su na primjer hijerarhijska stabla podataka.

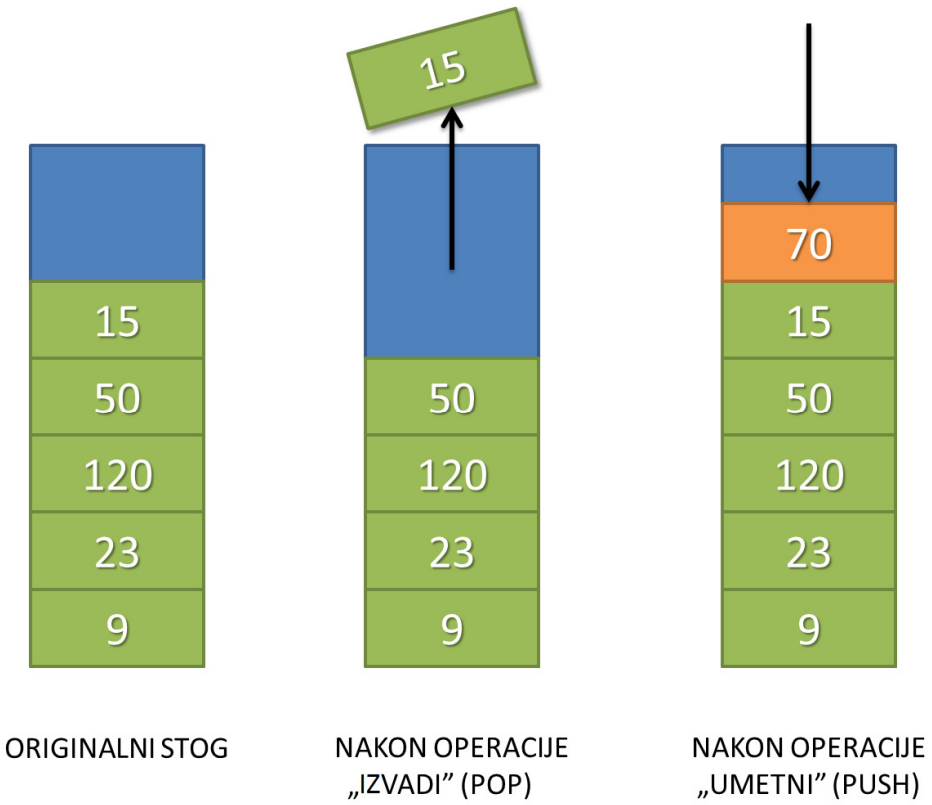
2.4. Kako rade rekurzivni programi?

Kod rada s rekurzivnim funkcijama koristi se struktura podataka koja se naziva stog (*engl.* stack) u koju se pohranjuju varijable rekurzivne funkcije.

No, što je to zapravo stog? Stog je apstraktni tip podataka koji služi za pohranu niza istovrsnih elemenata. Budući da stog radi po LIFO principu (*engl.* Last In First Out), posljednji pohranjeni podatak se uzima prvi u obradu [Drozdek, 2004., str. 137].

Na *slici 2.1.* može se vidjeti jedan jednostavan prikaz stoga te njegovih osnovnih funkcija „izvadi“ (*engl*. Pop) te „umetni“ (*engl*. Push).

Za što bolje razumijevanje rekurzivnih programa i funkcija treba znati što se ustvari događa kad funkcija poziva samu sebe. Kao što je već spomenuto, u tom procesu glavnu ulogu ima stog, to jest memorijski element iz kojeg se najprije učitava onaj sadržaj koji je u njega posljednji upisan.



Slika 2.1. Jednostavan prikaz stoga

Princip realizacije rekurzivne funkcije je sljedeći:

- rekurzivna funkcija poziva se sa određenom početnom vrijednošću varijable;
- računski proces unutar funkcije odvija se nesmetano sve do točke u kojoj dolazi do poziva istog potprograma sa izmijenjenom (najčešće smanjenom) vrijednošću argumenta;
- novi poziv iste funkcije dovodi do privremenog prekida izvršavanja tekućeg poziva rekurzivne funkcije. Pri tome se sve vrijednosti lokalnih varijabli spremaju na stog;
- ponavljanjem opisanog procesa stalno raste sadržaj stoga, sve dok ne stigne do krajnjeg uvjeta koji više ne zahtijeva rekurzivne pozive funkcije;
- povratkom s nižeg nivoa izvršavanja rekurzivne funkcije na viši nivo nastavlja se računski proces koji je bio prekinut, učitavaju se i obnavljaju vrijednosti lokalnih varijabli koje su bile pohranjene u stog;

- kad je dobiven pretposljednji rezultat rekurzivne funkcije, stog je doveden u početno stanje, izračunava se krajnja vrijednost rezultata i dostavlja pozivajućem programu.

Ovaj način rada troši dodatno vrijeme i memoriju na spremanje i očitavanje vrijednosti varijabli, a to se odražava na taj način da sporije dolazimo do rješenja.

Budući da je dokazano da se svako rekurzivno rješenje može napisati kao iterativno, rekurzija se preporuča koristiti samo onda kad se iterativnim pristupom teško dolazi do rješenja problema.

2.5. Usporedba rekurzivnih i iterativnih programa

Kao usporedba rekurzivnih i iterativnih programskih rješenja u nastavku se navodi primjer općepoznatog proračuna sume prirodnih brojeva od *1* do *n*.

Algoritam za sumu *n* brojeva je:

1. trivijalni slučaj: ako je *n==1*, *suma (n)* je jednaka 1
2. za ostale vrijednosti od *n*, *suma (n) = suma (n – 1)+ n*

U *tablici 2.1.* prikazana je usporedba rekurzivnog i iterativnog proračuna sume.

Tablica 2.1. Usporedba rekurzivnog i iterativnog proračuna sume

rekurzivni proračun sume	iterativni proračun sume

`int suma (int n){`
`  if (n == 1)`
`  return 1;`
`  else`
`  return suma(n – 1) + n;`
`}`
	`int suma (int n){`
	`  int sum=0;`
	`  while (n > 0){`
	`  sum += n;`
	`  n--;`
	`  }`
	`  return sum;`
	`}`

U oba se slučaja u svakom koraku vrši jedno logičko ispitivanje, jedno zbrajanje te jedno oduzimanje, no kod rekurzivnog rješenja se još gubi vrijeme i memorijski prostor za privremeni smještaj argumenata funkcije.

Može se zaključiti da, iako je rekurzivni kod kraći i „elegantniji“, isto tako je i neučinkovitiji (sporiji) te teži za analiziranje, no unatoč tome preporuča se korištenje rekurzivnih funkcija u onim slučajevima kad to odgovara prirodi problema.

2.6. Vrste rekurzivnih funkcija

Postoje dvije vrste rekurzije [Hartel, Muller, 1997., str. 51-74]:

- tail rekurzija,
- non - tail rekurzija.

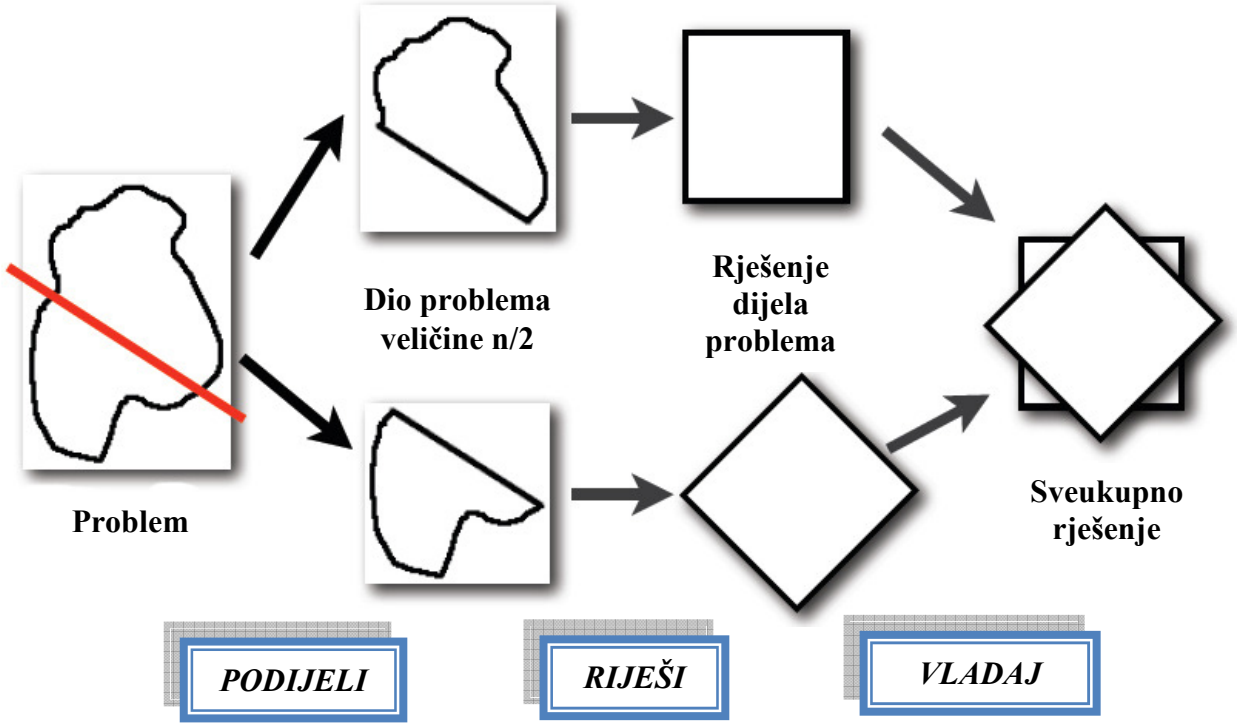
Tail rekurzija jest situacija u kojoj se rekurzija događa na repu (*engl.* tail) funkcije, što znači da nakon rekurzivnog poziva nema drugih instrukcija. Često je u takvim slučajevima rekurziju bolje zamijeniti iteracijom jer se time drastično smanjuje korištenje memorijskog prostora i povećava se efikasnost. Kao primjer *tail rekurzije* može se navesti brzo sortiranje (*engl*. Quick Sort). *Non - tail recursion* jest situacija u kojoj nakon poziva rekurzije postoje još druge instrukcije kao što je to slučaj kod sortiranja spajanjem (*engl.* Merge Sort).

3. Metoda *podijeli pa vladaj*

U prethodna dva poglavlja objašnjena je rekurzija na kojoj se zasniva metoda *podijeli pa vladaj*, dok će se u ovom poglavlju staviti naglasak na samu temu završnog rada.

Metoda *podijeli pa vladaj* posebna je algoritamska tehnika u kojoj se ulaz razbija na nekoliko dijelova te se potom problemi u svakom dijelu rješavaju rekurzivno, a zatim se kombiniraju rješenja ovih manjih problema u konačno rješenje [Kleinberg, Tardos, 2005., str. 197].

Na *slici 3.1.* može se vidjeti shema rada ove metode.



Slika 3.1. Shema metode *podijeli pa vladaj* [Izvor: Schwill, http://ddi.cs.uni-potsdam.de/]

U prvom koraku, ova metoda *teži* problem rastavlja na *lakše* probleme koji su slični polaznom problemu. Nakon toga rekurzija poziva samu sebe, ali ovaj put s tim *lakšim* problemom kao parametrom. Taj lakši problem dijeli se na još lakše i lakše, sve dok se ne dođe do osnovnog, trivijalnog problema koji znamo riješiti te tu dolazi do prekida rekurzije.

U većini slučajeva ovo je vrlo jednostavna, ali moćna metoda. Ona svojim konceptima znatno olakšava rješavanje kao i shvaćanje mnogih problema.

Metoda *podijeli pa vladaj* temelj je efikasnog algoritma za sve vrste problema, kao što su sortiranje (Quick Sort, sortiranje spajanjem), množenje velikih brojeva (npr. Karatsuba), Fourierove transformacije, itd. [Dasgupta, Papadimitriou, Vazirani, 2006., str. 55-90].

3.1. Kratka povijest metode *podijeli pa vladaj*

Jedna od prvih primjena ove izuzetno moćne metode bilo je binarno pretraživanje. Opis tog algoritma na računalima prvi se put pojavio 1946. godine u članku Johna Mauchlyja, američkog fizičara koji je dizajnirao ENIAC [Knuth, 1998, str. 422]. U tom je članku opisao ideju kojom bi se koristeći već sortirani niz elemenata znatno olakšalo pretraživanje. No, primjena ove metode seže još mnogo dalje u prošlost - nekoliko stoljeća prije Krista, Euklidov algoritam za računanje najvećeg zajedničkog djelitelja dvaju brojeva koristio je ovu metodu.

Jedan od značajnijih primjera jest algoritam koji je izumio Anatolii A. Karatsuba, ruski matematičar 1962. godine, a koji služi za množenje dva *n*-znamenkasta broja u poslovanju [Knuth, 1981, str. 279].

Postoji još mnogo primjera primjene ove metode koji će se navesti u daljnjem tekstu.

3.2. Vrste metode *podijeli pa vladaj*

S obzirom na način kako se dijeli originalni problem, postoje dvije vrste metode *podijeli pa vladaj*. To su [Puntambekar, 2008., str. 3, pog. 6]:

1. smanji pa vladaj
2. podijeli pa vladaj

Kod rješavanja problema podmetodom *smanji pa vladaj*, problem se rješava tako da se originalni problem smanji za neki faktor koji ovisno o problemu može biti konstantan ili varijabilan. Primjeri ove podmetode su *hanojski tornjevi*, *binarno pretraživanje*, itd.

Kod rješavanja problema podmetodom *podijeli pa vladaj*, problem se rješava tako da se originalni problem podijeli na *lakši* problem iste ili slične prirode te se tada pristupi rješavanju lakšeg problema. Primjeri ove podmetode su *Merge Sort*, *Quick Sort*, itd.

3.3. Implementacija

Podijeli pa vladaj algoritmi implementirani su kao rekurzivne funkcije, odnosno procedure. Kod objašnjavanja rekurzivnih funkcija napomenuto je da se kod rješavanja nekog dijela funkcije lokalne varijable spremaju na stog, te se može logički zaključiti da se kod izvođenja nekog od *podijeli pa vladaj* algoritama isto tako automatski poziva stog.

Za izvođenje algoritma potrebno je osigurati dovoljno memorije ili će u protivnom doći do preljeva (prepunjavanja) stoga (*engl. Stack overflow*). Premda kod implementacije vremenski efikasnog *podijeli pa vladaj* algoritma ne dolazi do velikih dubina rekurzije, a onda ni do čestog prepunjavanja stoga, kod implementacije ove metode nije potrebno riskirati. Rizik od prelijevanja stoga može se smanjiti ukoliko minimiziramo parametre i lokalne varijable rekurzivne funkcije ili pomoću definiranja eksplicitne stog strukture.

3.4. Složenost *podijeli pa vladaj* algoritama

Zbog jednostavnosti izračuna vremenske složenosti algoritama *podijeli pa vladaj* osmišljena je *Akra-Bazzi* metoda [Leighton, 1996., str. 1], poznatija još i kao *Akra-Bazzi* teorem, koja se koristi ukoliko originalni problem nije podijeljen na jednake dijelove. Ova je metoda izvedena prema takozvanom *Master teoremu* [Ibid, str. 1] koji pretpostavlja da se podjela originalnog problema izvršava na jednake dijelove.

3.4.1. Master teorem

Ovaj je teorem vrlo značajan u analizi algoritama te nudi jednostavna rješenja složenosti najčešće za rekurzivne algoritme. No, ovim se teoremom ne može odrediti svaka složenost. Kao što je rečeno, ovaj teorem pretpostavlja da su potproblemi originalnog problema jednake veličine.

Pretpostavimo problem koji se rekurzivno dijeli na nekoliko potproblema, a svaki problem je veličine $\frac{n}{b}$ (npr. operacije s binarnim stablom).

Složenost takvog problema tada se definira kao:

$$T_{a\lg oritam}(n) = a \cdot T \cdot \left(\frac{n}{b}\right) + f(n).$$

Ovaj se rekurzivni odnos može zamijeniti, odnosno biti dodatno proširen kako bi se dobio izraz za vremensku složenost cijelog algoritma.

Opći oblik ovog izračuna je:

$$T_{a\lg oritam}(n) = a \cdot T \cdot \left(\frac{n}{b}\right) + f(n), \text{ gdje je } a \geq 1, \ b \geq 1.$$

- n – predstavlja veličinu problema,
- a – predstavlja broj potproblema kod rekurzivnih poziva,
- $\frac{n}{b}$ – predstavlja veličinu svakog od potproblema (jednake veličine),
- $f(n)$ – predstavlja asimptotsku ocjenu složenosti problema bez rekurzivnih poziva, a uključuje složenosti podjele problema te spajanja rješenja potproblema.

Neki od popularnih algoritama koji za izračunavanje složenosti primjenjuju *Master teorem* su sortiranje spajanjem (*engl*. Merge Sort) i binarno pretraživanje (*engl*. Binary search).

3.4.2. Akra-Bazzi metoda

Već je ranije rečeno da je *Akra-Bazzi* metoda proširenje *Master teorema*. Ova metoda kreće od pretpostavke da se podjela originalnog problema ne izvršava na jednake dijelove. Zbog svoje se pretpostavke najčešće koristi kod *podijeli pa vladaj* algoritama.

Opći oblik:

$$T_{a\lg oritam}(x) = f(x) + \sum_{i=1}^{k} a_i T(b_i x + g_i(x)) \text{ za } x \geq x_0 .$$

Uvjeti za upotrebu su:

- dovoljan broj slučaja,
- a_i i b_i su konstante za svaki i,
- $a_i \geq 0$ za svaki i,
- $0 \leq b_i \geq 1$ za svaki i,
- $|f'(x)| \in O(x^c)$, c je konstanta,
- $|g_i(x)| \in O\left(\frac{x}{(\log x)^2}\right)$ za svaki i,
- x_0 je konstanta.

Akra-Bazzi metoda je korisnija, to jest ima veću upotrebu nego ostale metode za određivanje asimptotske ocjene složenosti jer pokriva širok spektar slučajeva. Osnovna namjena ove metode je određivanje složenosti mnogih *podijeli pa vladaj* algoritama.

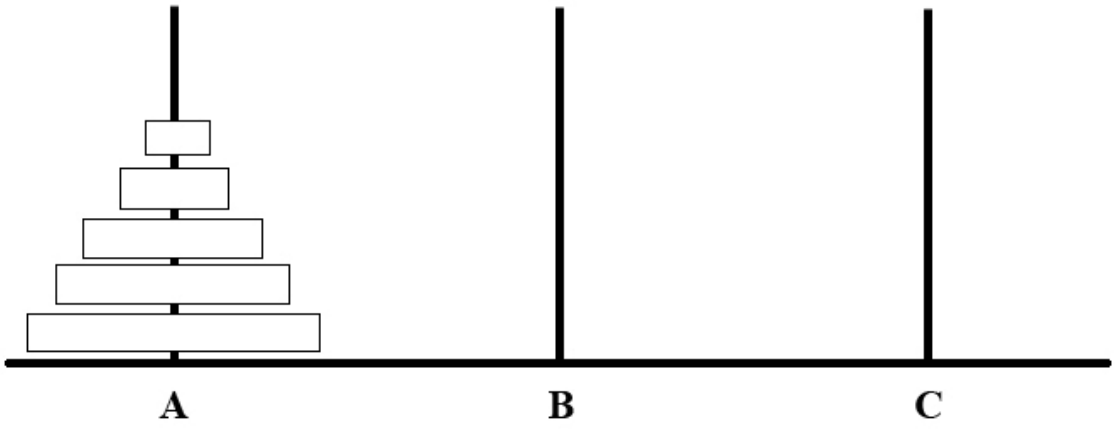
4. Pregled najpoznatijih primjera primjene metode

U ovom je poglavlju predstavljen pregled nekih najpoznatijih primjera primjene ove moćne metode kao što su hanojski tornjevi, sortiranje spajanjem (*engl.* Merge Sort), brzo sortiranje (*engl*. Quick Sort), binarno pretraživanje, itd.

4.1. Hanojski tornjevi

Hanojski tornjevi predstavljaju jedan od najpoznatijih rekurzivnih problema u računalnoj literaturi. Ovaj je zadatak izumio francuski matematičar Édouard Lucas 1883. godine prema legendi o vijetnamskom hramu u kojem se nalazi velika soba s tri istrošena stupa okruženih sa 64 zlatna diska [Chang, 2003., str. 88]. Hanojski su svećenici, djelujući izvan zapovjedništva dnevnog proročanstva, od tada pomicali te diskove u skladu s pravilima. Prema toj legendi, kad se povuče posljednji potez, to jest kad se riješi zagonetka, doći će do kraja svijeta. Postoje mnoga tumačenja vezana uz nastanak ove legende, od tvrdnji da ju je izmislio sam Lucas pa do činjenice da mu je samo poslužila kao inspiracija.

Problem je predstavljen na *slici 4.2.*



Slika 4.2. Hanojski tornjevi

Postoje tri štapa označena slovima *A*, *B* i *C*. Na prvi su štap nataknuti cilindrični diskovi različite veličine s rupom u sredini. Treba premjestiti sve diskove sa štapa *A* na štap *B* u redoslijedu kako se nalaze na štapu *A*, no pri prebacivanju diskova treba poštovati sljedeća pravila:

1. odjednom se smije pomicati samo jedan disk,
2. ne smije se stavljati veći disk na manji disk,
3. može se koristiti štap *C* za privremeni smještaj diskova, ali uz poštovanje prethodna dva pravila.

Ovaj problem moguće je definirati bilo kojim određenim brojem diskova. Broj poteza potrebnih da se riješi ovaj problem je 2^n-1, gdje je *n* broj diskova.

Prije nego što se krene s objašnjavanjem kako riješiti ovaj problem, potrebno je razjasniti zašto se hanojski tornjevi uopće smatraju rekurzivnim problemom.

U drugom je poglavlju rečeno da je nešto rekurzivno ako se može definirati *temeljni slučaj* i *pravilo rekurzije*. U ovom je problemu *temeljni slučaj* ako toranj sadrži samo jedan disk. Tad je rješenje vrlo jednostavno: taj se disk prebaci na ciljni toranj *B*. *Rekurzivno pravilo* bi bilo ako toranj sadrži *N* diskova. Pomicanje se diskova tada može izvesti u tri koraka:

1. Pomaknuti gornjih *N-1* diskova na pomoćni toranj *C*,
2. Preostali donji disk s tornja *A* pomaknuti na toranj *B*,
3. Zatim kulu od *N-1* diska s pomoćnog tornja *C* treba prebaciti na ciljni toranj *B*.

4.1.1. Jednostavno rješenje

Izmjenjivanje pokreta između najmanjeg i malo većeg diska jest jednostavno rješenje problema. Kod premještanja najmanjeg dijela uvijek se kreće u istom smjeru (na desnu stranu ukoliko je početan broj dijelova paran, odnosno na lijevu stranu ukoliko je početan broj dijelova neparan). Ukoliko ne postoji toranj kod odabranog smjera, pomiče se disk na suprotan kraj, ali se

nakon toga nastavi kretati u prije određenom smjeru. Na primjer, ako se počne s tri diska, pomiče se najmanji dio na suprotan kraj te se nakon toga nastavlja lijevo. Kad dođe trenutak za pomicanje malo većeg diska, postoji samo jedan ispravan pokret. Slijedeći ovakav način raspoređivanja diskova, trebalo bi problem riješiti najmanjim mogućim brojem koraka.

Ovo je rješenje jednostavno te ne zahtjeva poznavanje rekurzije. Međutim, rješavanje ovog problema korištenjem metode *podijeli pa vladaj* još je jednostavnije i elegantnije.

4.1.2. Rekurzivno rješenje

Kod ovakvog pristupa rješavanju ovog problema vrlo se lako prepoznaje metoda *podijeli pa vladaj* jer se problem razbija na manje probleme koji se dalje razbijaju na još manje probleme sve dok se ne dođe do rješenja.

U dosadašnjem su tekstu objašnjena pravila premještanja diskova, a sada slijedi prikaz funkcije koja izvršava gornja pravila. Funkcija će se zvati `pomakni_kulu`, a potrebni argumenti su: broj diskova koje treba pomaknuti, ime početnog tornja, ime ciljnog tornja te ime pomoćnog tornja.

```
void pomakni_kulu (int n, char A, char B, char C){
     if (n > 0){
          pomakni_kulu (n-1, A, C, B);     //1. pravilo
          pomakni_disk (A, B);             //2. pravilo
          pomakni_kulu (n-1, C, B, A);     //3. pravilo
     }
}
```

Za slučaj kad postoji samo jedan disk, funkcija `pomakni_kulu` ne izvršava ništa pa se u tom slučaju izvršava funkcija `pomakni_disk(A, B)` koja je definirana na sljedeći način:

```
void pomakni_disk (char sa_kule, char na_kulu){
     cout<<"\t\t"<<sa_kule<<" -> "<<na_kulu<<endl;
}
```

Ovakvo rekurzivno rješenje, mnogo je jednostavnije i elegantnije od nerekurzivnog rješenja. Teško da postoji neka druga metoda kojom bi se ovaj problem riješio na jednako efikasan način. Budući da se ne smije staviti veći disk povrh manjeg diska, potreban je eksponencijalni broj poteza, stoga je vremenska složenost ovakvog algoritma $T_{Hanoi}(n)=O(2^n)$.

4.2. Sortiranje spajanjem (Merge Sort)

Sortiranje spajanjem, poznatije još kao *Merge Sort*, jest algoritam sortiranja utemeljen na uspoređivanju. Algoritam sortiranja spajanjem prvi je osmislio američki matematičar mađarskog podrijetla John von Neumann 1945. godine [Knuth, 1998., str. 159].

Sortiranje spajanjem predstavlja jedan od najboljih primjera primjene metode *podijeli pa vladaj*. Ovo sortiranje radi na principu da se dani niz koji treba sortirati podijeli u dva jednaka dijela te sortira svaku polovicu rekurzivno, a zatim kombinira rezultate te ih se spaja u jedan sortirani niz.

Koraci izvođenja ovog algoritma:

1. ako niz ima nulu ili jedan element, tad je on već sortiran. Inače,
2. nesortirani niz se dijeli u dva podniza približno jednake dužine,
3. rekurzivno se sortira svaki podniz ponovnom primjenom algoritma sortiranja,
4. spajaju se dva sortirana podniza u jedan sortirani niz.

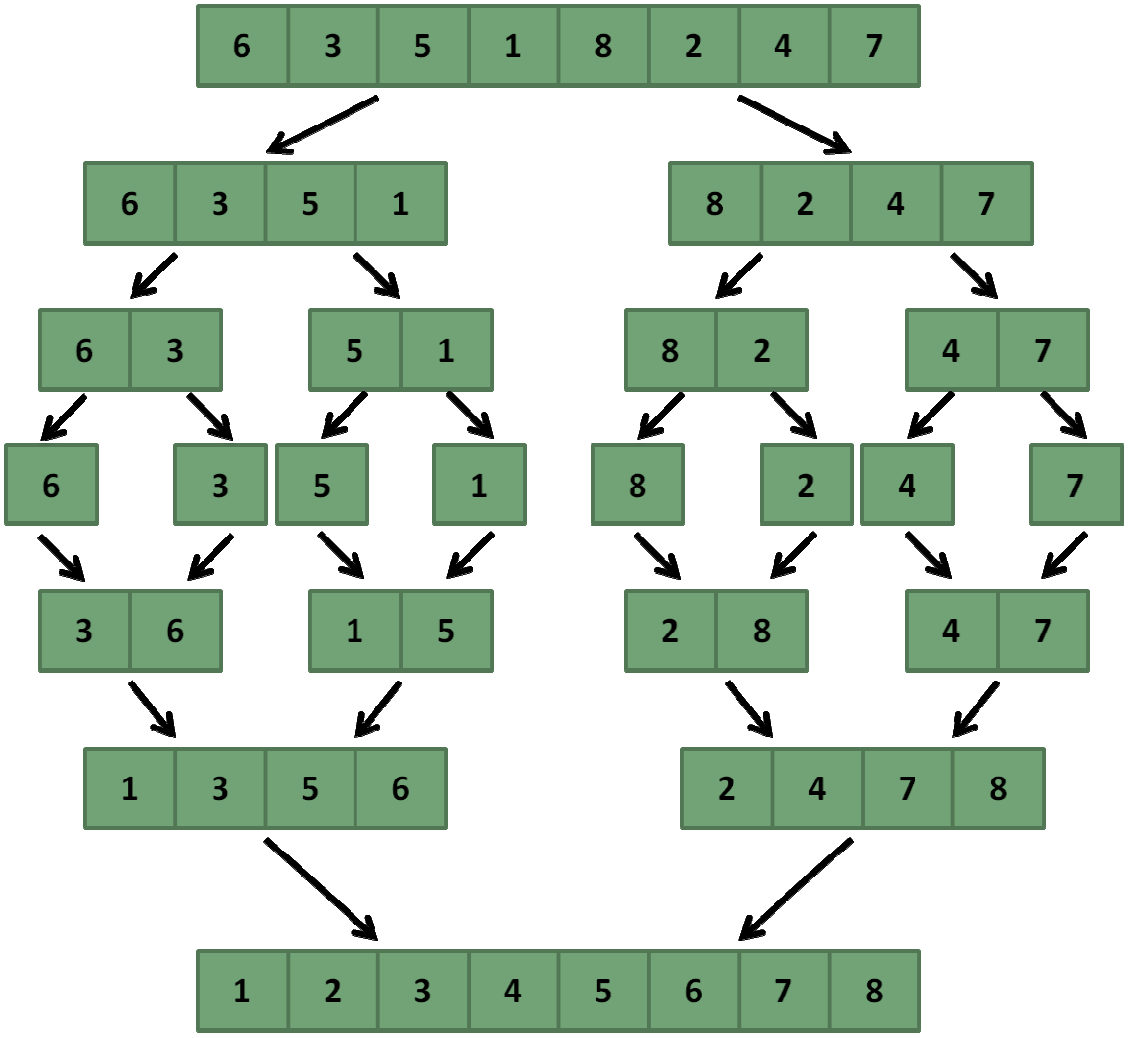
Opći pseudokod sortiranja spajanjem glasi:

```
MergeSort(mali, veliki)
      ako je (mali<veliki)
            srednji=(mali+veliki)/2
            MergeSort(mali, srednji)
            MergeSort(srednji+1, veliki)
            Merge(mali, srednji veliki)
kraj_MergeSort
```

No, osim same funkcije *MergeSort*, potrebna je i funkcija za spajanje čiji pesudokod glasi:

```
Merge(mali, srednji, veliki)
      h=i=mali
      j=srednji+1
      sve dok je ((h<=srednji) i (j<=veliki)) ponavljaj
            ako je (niz[h]<=niz[j])
                  b[i]=niz[h]
                  h++
            inače
                  b[i]=niz[j]
                  j++
      kraj ponavljanja
      ako je (h>srednji)
            za svaki k=j dok je k<=veliki ponavljaj
                  b[i]=niz[k]
                  i++
                  k++
            kraj ponavljanja
      inače
            za svaki k=h dok je k<=srednji ponavljaj
                  b[i]=niz[k]
                  i++
                  k++
            kraj ponavljanja
      za svaki k=mali dok je k<=veliki ponavljaj
            niz[k]=b[k]
      kraj ponavljanja
kraj_Merge
```

Na *slici 4.3.* prikazan je primjer sortiranja spajanjem.



Slika 4.3. Primjer sortiranja spajanjem

Kod sortiranja spajanjem kao i kod bilo kojeg algoritma koji koristi metodu *podijeli pa vladaj*, potreban je *temeljni slučaj* rekurzije. U sortiranju spajanjem to se događa kad je veličina ulaznog polja smanjena na veličinu 2 jer se tada zaustavlja rekurzija i sortiraju elementi međusobnim uspoređivanjem.

Ovaj algoritam koristi dva važna principa kojima se smanjuje vrijeme izvršavanja:

1. za sortiranje kratkog niza potreban je manji broj koraka nego za sortiranje dugačkog niza (osnova dijeljenja),
2. manje koraka je potrebno za konstrukciju sortiranog niza od dva sortirana podniza nego od dva nesortirana podniza (osnova spajanja).

4.2.1. Složenost Merge Sort algoritma

Sad slijedi izračun vremenske složenosti algoritma sortiranje spajanjem. Vremenska složenost sortiranja spajanjem u prosječnom kao i u najgorem mogućem slučaju iznosi $O(n\log(n))$.

Za analizu vremena izvođenja *Merge Sort* algoritma iskoristit će se sljedeći slučaj:

- podijeli ulaz u dva dijela jednake veličine, riješi dva potproblema rekurzivno, zatim kombiniraj ta rješenja u konačno rješenje trošeći samo vrijeme za inicijalno razdvajanje i finalno spajanje.

Vrijeme potrebno za sortiranje spajanjem niza duljine *n* je $T_{MergeSort}(n)$.

Pretpostavimo da je *n* paran broj. Tada algoritam troši *O(n)* vremena na podjelu ulaznog niza na dva dijela jednake veličine $\frac{n}{2}$ tako da pritom troši vrijeme $T_{MergeSort}\left(\frac{n}{2}\right)$ da riješi svaki dio, a na kraju provodi *O(n)* vremena za spajanje, to jest kombiniranje rješenja. Prema tome vrijeme potrebno za izvršavanje ovog algoritma bilo bi:

$$T_{MergeSort}(n) = 2T_{MergeSort}\left(\frac{n}{2}\right) + O(n).$$

Primjenjujući *Master teorem* dobiva se da je vremenska složenost $O(n\log(n))$.

4.3. Brzo sortiranje (Quick Sort)

Kao što mu i samo ime govori, brzo sortiranje, poznatije još kao Quick Sort, jedan je od najbržih poznatih algoritama za sortiranje i nije ga teško implementirati. 1960. godine osmislio ga je C. A. R. Hoare, američki znanstvenik [Sedgewick, 1983, str. 103]. Kao i prethodni algoritmi i ovaj je zasnovan na metodi *podijeli pa vladaj* na način da podijeli niz, odnosno listu, na dva podniza, odnosno dvije podliste. Quick Sort ima najbolji učinak ukoliko su podaci koji se nalaze u nizu, to jest koje je potrebno sortirati neuređeni, a niz veliki.

Iako je ovo jedan od najbržih algoritama za sortiranje, postoji jedan značajan nedostatak, a to je da je kao i kod svih rekurzivnih algoritama, potrebna dodatna memorija (najčešće stog) za spremanje lokalnih varijabli tokom izvođenja.

Koraci izvođenja ovog algoritma:

1. ako niz ima nulu ili jedan element, tada je on već sortiran. Inače,
2. niz se dijeli na dva dijela te se odabire stožerni element koji se još naziva *pivotom,*
3. niz se uredi na način da se svi elementi koji su manji od stožernog elementa stave ispred stožernog, a svi elementi koji su veći od stožernog stave iza stožernog,
4. rekurzivno se sortira podniz brojeva koji su manji od stožernog elementa, odnosno onih koji su veći.

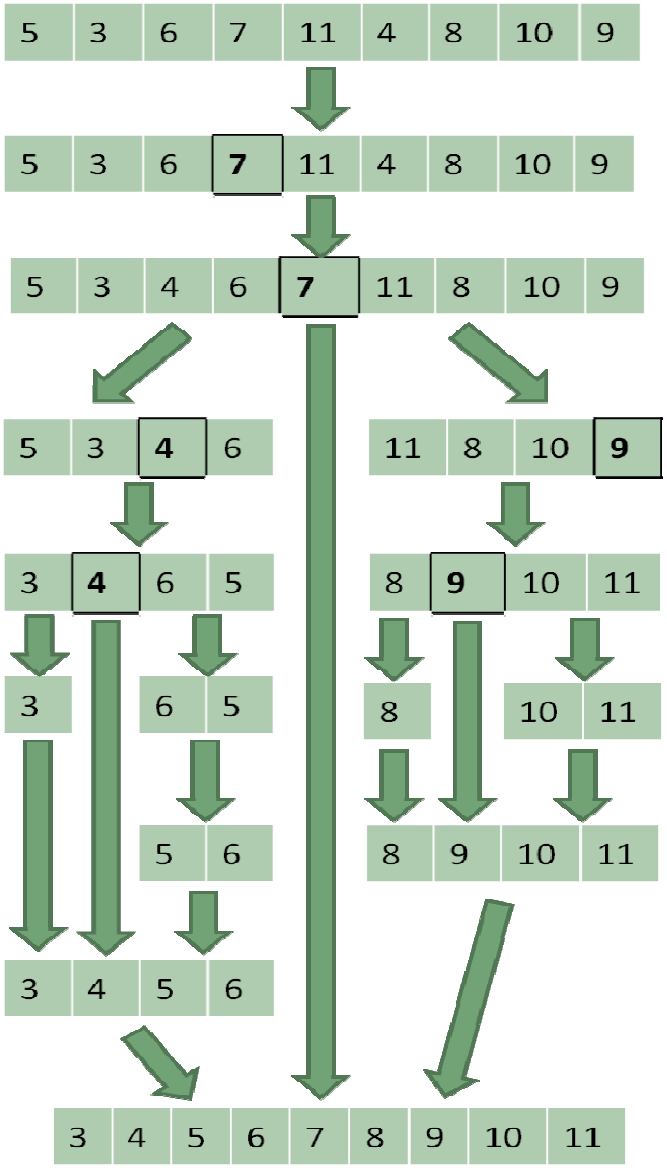
Za bolje shvaćanje koraka slijedi opći pseudokod brzog sortiranja:

```
QuickSort(lijevi, desni, niz[])
      i=lijevi, j=desni
      pivot=niz[(lijevi+desni)/2]
      ponavljanje
            dok je (niz[i]<pivot)
                  i=i+1
            dok je (pivot<niz[j])
                  j=j-1
            ako je (i<=j)
                  temp=niz[i]
                  niz[i]=niz[j]
                  niz[j]=temp
                  i=i+1
                  j=j-1
      sve dok je (i<=j)
            ako je (lijevi<j)
                  QuickSort(lijevi, j, niz)
            ako je (i<desni)
                  QuickSort(i, desni, niz)
kraj_QuickSort
```

Ispravnost ovog algoritma temelji se na sljedeća dva argumenta:

1. Kod svake iteracije svi se obrađeni elementi zasad nalaze u željenoj poziciji: prije stožernog elementa ako je element manji ili jednak u odnosu na vrijednost stožernog elementa, odnosno poslije stožernog elementa ukoliko je veći od njega.
2. Nakon svake iteracije (svakog rekurzivnog poziva), smanjuje se broj elemenata koji čekaju na obradu.

Na *slici 4.4.* može se vidjeti idealni slučaj izvođenja Quick Sort algoritma na primjeru niza slučajnih brojeva.



Slika 4.4. Primjer Quick Sort algoritma

Za stožerni element može biti odabran bilo koji element polja, no algoritam je najbrži kad je za stožerni element izabran srednji element polja. To je zato što se time dobivaju particije (podnizovi) vrlo slične veličine, a budući da se svaka particija dijeli na još dvije, vrlo se brzo dolazi do sortiranog niza.

4.3.1. Složenost Quick Sort algoritma

Dok je u najboljem i prosječnom slučaju vremenska složenost ovog algoritma $O(n\log(n))$, u najgorem je slučaju $O(n^2)$.

Najbolji slučaj je kao i kod sortiranja spajanjem – kad se niz podijeli na jednake dijelove (*Master teorem*).

$$T_{QuickSort}(n) = 2T_{QuickSort}\left(\frac{n}{2}\right) + O(n)$$

$$T_{QuickSort}(n) = 2\left(2T_{QuickSort}\left(\frac{n}{4}\right) + \frac{n}{2}\right) + O(n) = 4T_{QuickSort}\left(\frac{n}{4}\right) + 2\cdot O(n)$$

$$T_{QuickSort}(n) = c\log n + \sum_{k=0}^{\log n} n = c\log n + n(\log n + 1) \quad \Rightarrow$$

$$T_{QuickSort}(n) = O(n\log n).$$

Najgori slučaj (kad je loša podjela niza prema pivotu):

$$T_{QuickSort}(n) = T_{QuickSort}(n-1) + T(1) + O(n) = T_{QuickSort}(n-2) + T(1) + T_{QuickSort}(n-1) + T(1) + O(n) =$$

$$= nT(1) + \sum_{k=1}^{n} k \quad \Rightarrow$$

$$T_{QuickSort}(n) = O(n^2).$$

4.4. Binarno pretraživanje (Binary Search)

Binarno je pretraživanje algoritam za pretragu nekog niza ukoliko je taj niz već uzlazno sortiran. Ovo je jedan od najbržih načina pretraživanja niza koji je najbolje koristiti kod velikih

nizova. On uzima srednji element sortiranog niza, dijeli niz na pola te, ukoliko je tražena vrijednost manja ili jednaka od tog srednjeg elementa, odbacuje desni podniz gdje se nalaze vrijednosti veće od srednjeg elementa te rekurzivno ponavlja postupak na podnizu gdje se nalaze vrijednosti manje od srednjeg elementa.

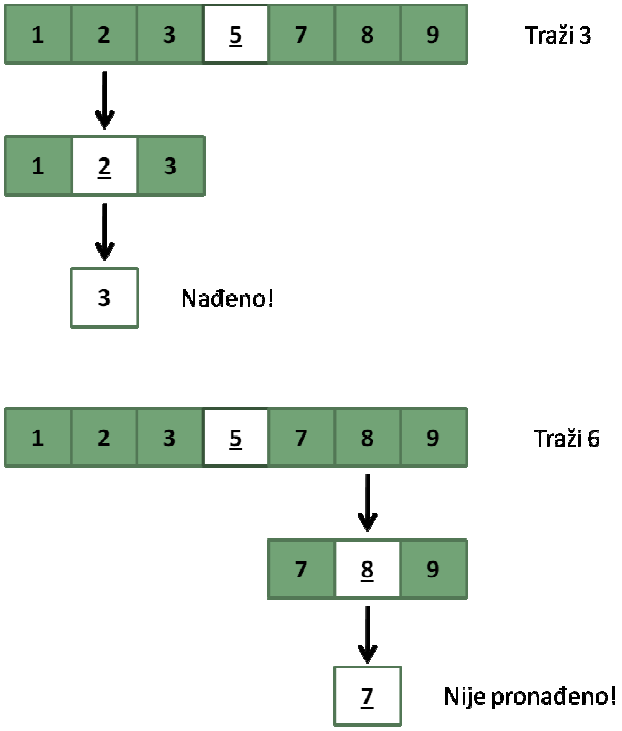
Koraci kod izvršavanja binarnog pretraživanja:

- pronalazi se srednji element,
- prema uvjetu se odbacuje jedna polovica niza,
- pretražuje se druga polovica rekurzivno sve dok se traženi element ne pronađe ili dok ne ostane niti jedan element za pretraživanje.

Opći pseudokod binarnog pretraživanja glasi:

```
BinarySearch(niz[], pocetak, kraj, broj)
     ako je (pocetak<=kraj)
          srednji=(pocetak+kraj)/2
          ako je (niz[srednji]==broj)
               ispis: "Broj je pronađen!"
          ako je (broj<niz[srednji])
               kraj=srednji-1
               BinarySearch(niz, pocetak, kraj, broj);
          inače
               pocetak=srednji+1
               BinarySearch(niz, pocetak, kraj, broj);
kraj_BinarySearch
```

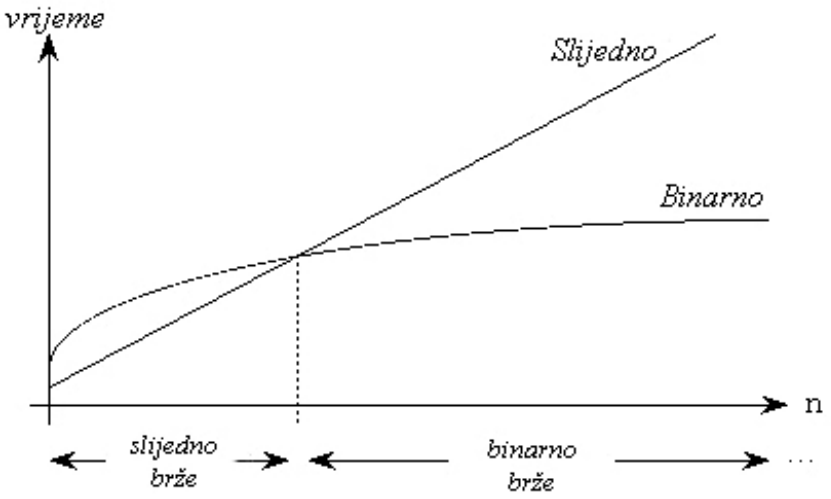
Na *slici 4.5.* prikazan je primjer binarnog pretraživanja.



Slika 4.5. Primjer binarnog pretraživanja

Na *slici 4.6.* prikazana je usporedba binarnog i slijednog pretraživanja.

Na slici se može primijetiti kako je slijedno pretraživanje samo prvih nekoliko trenutaka bolje od binarnog pretraživanja, a tada vrijeme utrošeno na slijedno pretraživanje linearno raste dok vrijeme binarnog pretraživanja ostaje isto.



Slika 4.6. Usporedba slijednog i binarnog pretraživanja [Izvor: Drysdale, http://www.cs.dartmouth.edu]

Ovaj se algoritam može implementirati na više različitih načina, no najelegantnije kao i najjednostavnije rješenje jest ono rekurzivno.

4.4.1. Složenost

U najgorem slučaju vremenska je složenost ovog algoritma $O(n\log(n))$, u najboljem slučaju je *O(1)*, dok je u prosječnom slučaju $O(\log n)$.

Prosječni slučaj:

$$T_{BinarySearch}(n) = 2T_{BinarySearch}\left(\frac{n}{2}\right) + O(1)$$

Kad se na to primijeni *Master teorem* gdje je $f(n) = n^c$ dobiva se:

$$T_{BinarySearch}(n) = O(\log n).$$

5. Primjena metode u rješavanju modernih problema

U prethodnim je poglavljima nekoliko puta naglašeno da se metoda *podijeli pa vladaj* koristi u rješavanju raznovrsnih praktičnih problema. Naime, uvijek je dobro neki kompleksan problem podijeliti na niz manjih čijim se rješavanjem u konačnici dolazi do rješenja početnog problema. Tako je ova metoda dobila svoju primjenu i u rješavanju nekih modernih problema kao što je analiza socijalnih mreža koja će biti opisana u nastavku.

Socijalne mreže su računalno potpomognute mreže ljudi gdje oni kroz interakciju razmjenjuju znanja i iskustva ili se bave provođenjem zabavnih aktivnosti [Orehovački, Konecki, Radošević, 2008., str. 269 - 273]. Pojavom Web 2.0 paradigme [O'Reilly, 2005.] temeljne karakteristike socijalnih mreža dobile su svoju online dimenziju. Od popularnijih socijalnih mreža potrebno je izdvojiti *Facebook* [http://www.facebook.com], *Twitter* [http://twitter.com], *MySpace* [http://www.myspace.com], *Hi5* [http://hi5.com/friend/displayHomePage.do] te *Friendster* [http://www.friendster.com].

Sve većim korištenjem socijalnih mreža došlo je do potrebe za njihovim analiziranjem. Postoje brojni razlozi i zahtjevi za njihovu analizu kao što su: socijalni, ekonomski, društveni, politički, itd. Nepredviđeni rast ovih mreža predstavlja problem kod njihovog dizajniranja i održavanja kao i pretrage. Jedan od načina rješavanja ovog problema jest pokušaj rješavanja metodom *podijeli pa vladaj* i to tako da se velika mreža podijeli na manje istovrsnih dijelova rekurzivno prema nekim svojstvima, a potom se svaki od tih dijelova dodjeli nekom serveru koji brine o povezanosti tako dobivenih dijelova. Time se znatno olakšava upravljanje socijalnim mrežama kao i njihovo održavanje te pretraživanje po određenim svojstvima [Pujol, Erramilli, Rodriguez, 2009., str. 1-2].

6. ZAKLJUČAK

U ovome završnom radu prikazani su i objašnjeni glavni koncepti kao i primjena metode *podijeli pa vladaj* te su implementirani neki od najpoznatijih primjena ove metode u programskom jeziku *C++*.

Za razumijevanje ove korisne metode bilo je potrebno shvatiti što je rekurzija te kako ona radi. Rekurzija u programiranju predstavlja koncept koji omogućuje iterativno ponavljanje određenih akcija unutar funkcije ili procedure na način da određeni dio koda inicira ponovno pozivanje te iste funkcije ili procedure. Rekurzivne funkcije pojednostavljuju programsko rješenje, no nedostatak im je što zauzimaju više memorijskog prostora te im je potrebno duže vrijeme za izvođenje. Korištenje rekurzivnih funkcija preporuča se onda kad je problem već sam po sebi rekurzivan.

Na rekurziji se zasniva metoda *podijeli pa vladaj*. To je posebna algoritamska tehnika u kojoj se ulaz razbija na nekoliko dijelova te se potom problemi u svakom dijelu rješavaju rekurzivno, a zatim se kombiniraju rješenja ovih manjih problema u konačno rješenje. Ovo je jedna jednostavna metoda koja je pronašla svoju primjenu u mnogim područjima kao što su razne vrste sortiranja, pretraživanje, množenje velikih brojeva, Fourierove transformacije, itd. Osim primjene kod nekih dobro poznatih problema, ova se metoda primjenjuje i u nekim novijim, ne još potpuno istraženim područjima kao što je analiza socijalnih mreža. Budući da se ova metoda zasniva na rekurziji ima isti nedostatak kao i sama rekurzija – zauzimanje više memorijskog prostora. No, metoda *podijeli pa vladaj* znatno olakšava implementaciju programskih rješenja te joj je glavna prednost jednostavnost te često i efikasnost.

Proučavanjem ove metode nameće se zaključak da je to jedna od najefikasnijih metoda za rješavanje raznih tipova problemskih situacija zbog svojih jasno definiranih pravila korištenja te kvalitetnih i brzih rješenja.

7. Literatura

1. Chang, S. K. (2003) *Data structures and algorithms*, Singapur: World Scientific Publishing Company.
2. Dasgupta, S., Papadimitriou, C. H., Vazirani, U. V. (2006) *Algorithms*, New York: McGraw-Hill Companies.
3. Drozdek, A. (2004) *Data Structures and Algorithms in C++*, *Third Edition*, Boston: Course Technology.
4. Drysdale, S., (2000) *Comparing linear search and binary search*, http://www.cs.dartmouth.edu/farid/teaching/cs15/cs5/lectures/0519/0519.html, učitano 30.06.2010.
5. Hartel, P., Muller, H. (1997) *Functional C*, Massachusetts: Addison-Wesley Professional.
6. Kleinberg, J., Tardos, E. (2005) *Algorithm Design*, Massachusetts: Addison-Wesley Professional.
7. Knuth, D. E. (1998) *The Art of Computer Programming*, *Volumes 1-3*, *Second Edition*, Massachusetts: Addison-Wesley Professional.
8. Knuth, D. E. (1981) *The Art of Computer Programming*, *Volume 2*, *Second Edition*, Massachusetts: Addison-Wesley Professional.
9. Lee, K. D. (2008) *Programming Languages, An Active Learning Approach*, New York: Springer.
10. Leighton, T. (1996) *Notes on Better Master Theorems for Divide-and-Conquer Recurrences*, http://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf, učitano 29.06.2010.
11. Metcalf, M., Reid, J., Cohen, M. (2004) *Fortran 95/2003 explained*, Oxford: Oxford University Press.
12. Orehovački, T., Konecki, M., Radošević, D. (2008), *Web 2.0 technologies in university education*, Zbornik radova 31. međunarodnog skupa za informacijsku i komunikacijsku tehnologiju, elektroniku i mikroelektroniku, MIPRO: Opatija.
13. O'Reilly, T., *What Is Web 2.0*, http://oreilly.com/web2/archive/what-is-web-20.html, učitano 01.07.2010.

14. Pujol, J. M., Erramilli, V., Rodriguez, P. (2009), *Divide and Conquer: Partitioning Online Social Networks*, http://arxiv.org/pdf/0905.4918, učitano 30.06.2010.

15. Puntambekar, A. A. (2008) *Analysis And Design Of Algorithms*, Pune: Technical Publications.

16. Schwill, A. (2006) *Algorithmenmuster*, http://ddi.cs.uni-potsdam.de/Lehre/GdP2/Folien/Kapitel5-2006.pdf, učitano 23.06.2010.

17. Sedgewick, R. (1983) *Algorithms*, Massachusetts: Addison Wesley Professional.

8. Popis slika i tablica

Popis tablica:

TABLICA 2.1. USPOREDBA REKURZIVNOG I ITERATIVNOG PRORAČUNA SUME 6

Popis slika:

SLIKA 2.1. JEDNOSTAVAN PRIKAZ STOGA 5
SLIKA 3.1. SHEMA METODE *PODIJELI PA VLADAJ* 8
SLIKA 4.2. HANOJSKI TORNJEVI 13
SLIKA 4.3. PRIMJER SORTIRANJA SPAJANJEM 18
SLIKA 4.4. PRIMJER QUICK SORT ALGORITMA 21
SLIKA 4.5. PRIMJER BINARNOG PRETRAŽIVANJA 24
SLIKA 4.6. USPOREDBA SLIJEDNOG I BINARNOG PRETRAŽIVANJA 24

9. Prilozi

TornjeviHanoi.cpp

MergeSort.cpp

QuickSort.cpp

BinarySearch.cpp

Uz spomenute priloge na kraju završnog rada priložen je CD s digitalnom verzijom rada kao i implementacijama.

9.1. TornjeviHanoi.cpp

```
#include <iostream>
using namespace std;

void pomakni_disk (char sa_kule, char na_kulu){
     cout<<"\t\t\t"<<sa_kule<<" -> "<<na_kulu<<endl;
}//pomakni_disk

void pomakni_kulu (int n, char A, char B, char C){
     if (n > 0){
        pomakni_kulu (n-1, A, C, B);
        pomakni_disk (A, B);
        pomakni_kulu (n-1, C, B, A);
    }//if
}//pomakni_kulu

int main (){
   int n;
   cout<<"\n\n\tBroj diskova: ";
   cin>>n;
   system("cls");
   cout<<"\n\n\tSlijed poteza za "<<n<<" diskova: \n\n";
   pomakni_kulu (n, 'A','B','C');
   cout<<"\n\n";
   system ("pause");
   return 0;
}//main
```

9.2. MergeSort.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;
int niz[1000];

void Merge(int mali, int srednji, int veliki){
     int h, i, j, k, b[1000];
     h=i=mali;
     j=srednji+1;
     while((h<=srednji) && (j<=veliki)){
          if(niz[h]<=niz[j]){
               b[i]=niz[h];
               h++;
          }else{
               b[i]=niz[j];
               j++;
          }//if else
          i++;
      }//while
          if(h>srednji){
               for(k=j; k<=veliki; k++){
                    b[i]=niz[k];
                    i++;
               }//for
          }else{
               for(k=h; k<=srednji; k++){
                    b[i]=niz[k];
                    i++;
               }//for
          }//if else

     for(k=mali; k<=veliki; k++)
          niz[k]=b[k];
}//Merge

void MergeSort(int mali, int veliki){
     int srednji;
     if (mali<veliki){
        srednji=(mali+veliki)/2;
        MergeSort(mali, srednji);
        MergeSort(srednji+1, veliki);
        Merge(mali, srednji, veliki);
     }//if
}//MergeSort
```

```
int main(){
      int broj, i;
      cout<<"\n\nUnesi broj elemenata u nizu: ";
      cin>>broj;
      cout<<"\n\nElementi niza su: \n\n";
      //niz=new int[broj];
      for(i=1; i<=broj; i++){
            niz[i]=rand()%1000+1;
            cout<<setw(5)<<niz[i];
      }//for
      cout<<"\n\n";
      MergeSort(1, broj);
      cout<<"\n\nSortirani niz: \n\n";
      for(i=1; i<=broj; i++){
         cout<<setw(5)<<niz[i];
      }//for
      //delete [] niz;
      cout<<"\n\n\n";
      system("pause");
      return 0;
}//main
```

9.3. QuickSort.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;
int niz[1000];

void QuickSort(int lijevi, int desni, int niz[]){
      int i=lijevi;
      int j=desni;
      int pivot=niz[(lijevi+desni)/2];
      while (i<=j){
            while(niz[i]<pivot)
                  i=i+1;
            while(pivot<niz[j])
                  j=j-1;
            if(i<=j){
                  int temp=niz[i];
                  niz[i]=niz[j];
                  niz[j]=temp;
                  i=i+1;
                  j=j-1;
            }//if
      }//while
      if (lijevi < j) QuickSort(lijevi, j, niz);
      if (i < desni) QuickSort(i, desni, niz);
}//QuickSort

int main(){
      int broj, i;
      cout<<"\n\nUnesi broj elemenata u nizu: ";
      cin>>broj;
      cout<<"\n\nElementi niza su: \n\n";
      for(i=1; i<=broj; i++){
            niz[i]=rand()%1000+1;
            cout<<setw(5)<<niz[i];
      }//for
      cout<<"\n\n";
      QuickSort(1, broj, niz);
      cout<<"\n\nSortirani niz: \n\n";
      for(i=1; i<=broj; i++){
         cout<<setw(5)<<niz[i];
      }//for
      cout<<"\n\n\n";
      system("pause");
      return 0;
}//main
```

9.4. BinarySearch.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;

void BinarySearch(int niz[15], int pocetak, int kraj, int broj){
      int srednji, nasao;
      if(pocetak<=kraj){
            srednji=(pocetak+kraj)/2;
            if(niz[srednji]==broj){
                  cout<<"\n\t\tBroj pronaden!\n";
                  nasao=1;
            }//if2
            if(broj<niz[srednji]){
                  kraj=srednji-1;
                  BinarySearch(niz, pocetak, kraj, broj);
            }else{
                  pocetak=srednji+1;
                  BinarySearch(niz, pocetak, kraj, broj);
            }//if else
      }else if (pocetak>kraj && nasao==0){
             cout<<"\n\t\tBroj nije pronaden!\n";
    }//if1
}//BinarySearch

int main(){
      int niz[]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
      int i, broj;
      cout<<"\n\nElementi niza su: \n\n";
      for(i=0;i<15;i++){
            cout<<setw(5)<<niz[i];
      }
      cout<<"\n\n\tTrazi: ";
      cin>>broj;

      BinarySearch(niz, 0, 15, broj);
      cout<<"\n\n";
      system("pause");
      return 0;
}//main
```