

## 13.1 Rekurzivne funkcije

U programiranju *rekurzija* nastaje kada funkcija poziva samu sebe direktno ili indirektno. *Indirektna rekurzija* nastaje kada jedna funkcija poziva drugu funkciju, iz koje se ponovo poziva pozivna funkcija.

U matematici se često proračun neke funkcije  $f(n)$  definira pomoću rekurzije.

Koristi se pravilo:

1. postavi  $f(0)$  "temeljni slučaj"
2. računaj  $f(n)$  pomoću  $f(k)$  za  $k < n$  "pravilo rekurzije"
- 3.

Primjerice, rekurzivno se može izračunati  $n!$  ( $n$ -faktorijela), jer vrijedi

za  $n=0$ :  $0! = 1$ ; (temeljno rekurzije)  
za  $n>0$ :  $n! = n * (n-1)!$  (rekurzivno pravilo)

pa se sve vrijednosti mogu izračunatu pomoću gornjeg rekurzivnog pravila:

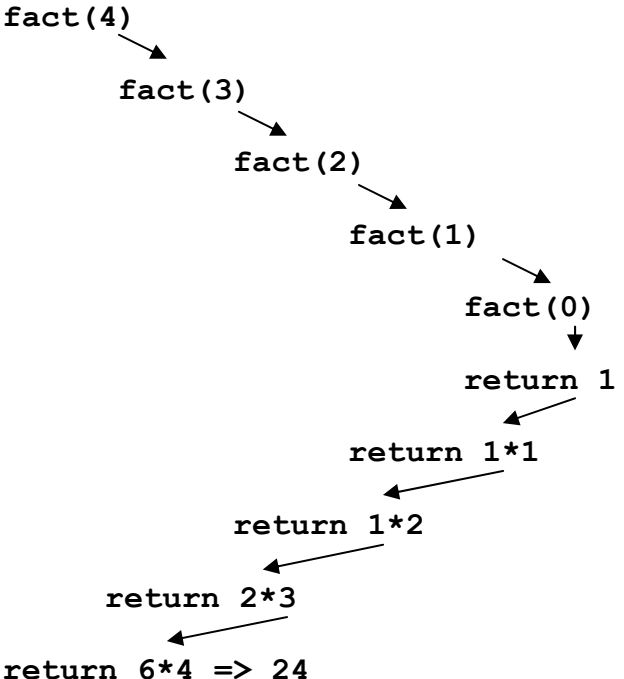
$1! = 1 * 0! = 1$   
 $2! = 2 * 1! = 2$   
 $3! = 3 * 2! = 6 \quad \dots \text{ itd.}$

Ovo razmatranje zaključujemo rekurzivnom definicijom funkcije za proračun  $n!$ :

```
int factorial(int n)
{
    if (n == 0)        return 1;
    else               return factorial(n-1) * n;
}
```

### 13.3 Zauzeće izvršnog stoga kod poziva rekurzivnih funkcija

Primjer izlaza za slučaj proračuna 4!:

izvršenje poziva funkcije fact(4)	stanje memorije koja se koristi za prijenos argumenata funkcije (izvršni stog)
 <pre>graph TD     fact4[<b>fact(4)</b>] --&gt; fact3[<b>fact(3)</b>]     fact3 --&gt; fact2[<b>fact(2)</b>]     fact2 --&gt; fact1[<b>fact(1)</b>]     fact1 --&gt; fact0[<b>fact(0)</b>]     fact0 --&gt; r1[<b>return 1</b>]     r1 --&gt; r2[<b>return 1*1</b>]     r2 --&gt; r3[<b>return 1*2</b>]     r3 --&gt; r4[<b>return 2*3</b>]     r4 --&gt; r5[<b>return 6*4 =&gt; 24</b>]</pre>	<pre>... 4 ... 4 3 ... 4 3 2 ... 4 3 2 1 ... 4 3 2 1 0 ... 4 3 2 1 ... 4 3 2 ... 4 3 ... 4 ...</pre>

Slika 1. Redoslijed izvršenja rekurzivne funkcije `fact(4)`

U programu fact-stog.cpp realizirana je funkcija factorial koristeći simulaciju izvršnog stoga. Program ispisuje stanje na stogu u svakom koraku rekurzije. Cilj je vidjeti kako rekurzivne funkcije koriste izvršni stog.

```
//Program: fakt-stog.c
// Proračun n!. Vrijednost od n [0,13]
#include <iostream>
#include <vector>
using namespace std;

vector<int> S;          // simulator izvršnog stoga

int Reg;               // registar za rezultat funkcija

void printstog()
{
    cout <<"Stog: ";
    for(int i=0; i<S.size(); i++)
        cout << S[i] <<" ";
    cout << endl;
}

void push(int n)       {S.push_back(n); printstog(); }

int & top(int n=0)     {return S[S.size()-1+n];}

void pop()             {S.pop_back(); printstog();}
```

```

void factorial()
{
    int n = top();          // n je lokalna varijabla
    if (n == 0)
        Reg = 1;           // rezultat za n==1
    else
    {
        push(n-1);          // poziv za arg = n-1
        factorial();
        pop();
        Reg = n * Reg;      // vraca rezultat
    }
}

int main()
{
    int n, result;
    cout << "Unesite broj unutar intervala [0,13]\n";
    cin >> n;

    if((n < 0) || (n > 13)) exit(1);

    push(n);
    factorial();
    result = Reg;
    pop();
    cout << "Vrijednost " << n
         << "! iznosi: " << result << endl;
    return 0;
}

```

}

## 13.4 Usporedba rekurzivnih i iterativnih programa

Za usporedbu rekurzivnih i iterativnih programskih rješenja dati ćemo primjer potprograma za proračun sume prirodnih brojeva 1,2,..n:

Rekurzivni algoritam za sumu n brojeva je:

1. trivijalni slučaj: ako je  $n==1$  suma(n) je jednaka 1
2. za ostale vrijednosti od n,  $\text{suma}(n) = \text{suma}(n-1)+n$

rekurzivni proračun sume 1,2,..n	iterativni proračun sume 1,2,..n
<pre>unsigned suma( unsigned n) {     if (n == 1)         return 1;     else         return suma(n - 1) + n; }</pre>	<pre>unsigned suma(unsigned n) {     unsigned sum=0;     while (n &gt; 0) {         sum += n;         n--;     }     return sum; }</pre>

U obje funkcije u svakom se koraku vrši jedno logičko ispitivanje, jedno zbrajanje i jedno oduzimanje. U rekurzivnoj funkciji se još gubi vrijeme i memorijski prostor za privremeni smještaj argumenata funkcije.

Usporedbom ova dva primjera vidljivo je da iterativni programi daju brži proračun i manje zauzimaju memoriju. To je čest slučaj.

## 13.5 Metoda - podijeli pa vladaj (Divide and Conquer)

Opći princip metode ja da se problem podijeli u više manjih problema.

Primjer: Približan proračun drugog korijena broja  $n$

Metoda: odredi vrijednost  $x$  za kojeg je  $(n - x^2)$  približno jednako nuli.

### 1) Temeljne pretpostavke:

Točno rješenje se nalazi unutar intervala  $[d, g]$ . Primjerice, sigurno je da se rješenje nalazi u intervalu  $[0, n]$  ako je  $n > 1$ , odnosno u intervalu  $[0, 1]$  ako je  $n < 1$ . Interval može biti i uži ako smo sigurni da obuhvaća točno rješenje.

Uobičajeno se uzima da je vrijednost od  $x$  u sredini intervala, tj.

$$x = (d + g) / 2.$$

Vrijednost  $x$  je blizu točnog rješenja ako je širina intervala manja od neke po volji odabrane vrijednosti *epsilon* (npr. 0,000001), tj. ako je  $g - d < \epsilon$ .

Ako je širina intervala veća od *epsilon*, do rješenje dolazimo koristeći rekurziju preme sljedećem pravilu:

### 2) Pravilo rekurzije:

Ako je  $n < x^2$  rješenje tražimo u intervalu  $[d, x]$ , u suprotnom tražimo u intervalu  $[x, g]$ .

---

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

#define EPSILON 0.000001

double sqrt_rek(double n, double d, double g)
{
    double x;
    x = (d + g) / 2.0;
    if (g - d < EPSILON)
        return x;
    else if (n < x*x )
        return sqrt_rek(n, d, x);
    else
        return sqrt_rek(n, x, g);
}
```



## 13.6 Binarno pretraživanje niza

Zadan je niz cijelih brojeva  $a[n]$  kojem su elementi sortirani od manje prema većoj vrijednosti, tj.

$$a[i-1] < a[i], \quad \text{za } i=1, \dots, n-1$$

Želimo odrediti da li se u ovom nizu nalazi element vrijednosti  $x$ . U tu svrhu koristit ćemo funkciju `binSearch` koja daje indeks  $i$ , elementa  $a[i]$ , kojem je vrijednost jednaka traženoj vrijednosti  $x$ . Ako vrijednost od  $x$  nije jednaka ni jednom elementu niza funkcija `binSearch` vraća negativnu vrijednost  $-1$ . Također, uzet ćemo da se može ispitati i samo dio niza, od nekog donjeg indeksa  $d$  do gornjeg indeksa  $g$ . Funkcija `binSearch`, dakle, mora imati argumente: niz koji se pretražuje, traženu vrijednost  $x$ , te donju i gornju granicu indeksa niza unutar koje se vrši traženje zadane vrijednosti.

```
int binSearch( int a[], int x, int d, int g );
```

Problem definiramo rekurzivno na sljedeći način:

### 1. Temeljne pretpostavke:

Ako u nizu  $a[i]$  postoji element jednak traženoj vrijednosti  $x$ , njegov indeks je iz intervala  $[d, g]$ , gdje mora biti istinito  $g \geq d$ . Trivijalni slučaj je za  $d=0$ ,  $g=n-1$ , koji obuhvaća cijeli niz.

Razmatramo element niza indeksa  $i = (g+d)/2$ . (dijelimo niz na dva podniza)

Ako je  $a[i]=x$ , pronađen je traženi element niza i funkcija vraća indeks  $i$ .

### 2. Pravilo rekurzije:

Ako je  $a[i] < x$

    rješenje tražimo u intervalu  $[i+1, g]$ ,

inače

    je u intervalu  $[d, i-1]$ .

```

int binSearch( int a[],int x, int d, int g)
{
    int i;
    if (d > g) return -1;
    i = (d + g)/ 2;
    if (a[i] == x)
        return i;
    if (a[i] < x)
        return binSearch( a, x, i + 1, g);
    else
        return binSearch( a, x, d, i - 1);
}

```

**Ilustrirajmo proces traženja vrijednosti x=23 u nizu od 14 elemenata**

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	3	5	6	8	9	12	23	26	27	31	34	42
d						i			g				
1	2	3	5	6	8	9	12	23	26	27	31	34	42
d								i		g			
1	2	3	5	6	8	9	12	23	26	27	31	34	42
d							i		g				

- 1.korak: d=0,      g=13,      i=6,      a[6]<23  
 2.korak: d=i+1=7, g=14,      i=10, a[10]>23  
 3.korak: d=7,      g=i-1=9, i=8,      a[8]==23

## 13.7 Iteracija kao specijalni slučaj rekurzije

Razmotrimo funkciju `petlja()` unutar koje se izvršava `do-while` petlja:

```
void petlja()
{
    do
        iskaz
    while (e);
}
```

Ova funkcija je semantički jednaka sljedećoj rekurzivnoj funkciji:

```
void petlja()
{
    iskaz
    if (e) petlja();
}
```

Rekurzivni poziv se vrši na kraju (ili na repu) tijela funkcije, stoga se ovaj tip rekurzije naziva "tail" rekurzija ili "rekurzija na repu". Neki optimizirajući kompajleri mogu prepoznati ovakav oblik rekurzivne funkcije i transformirati rekurzivno tijelo funkcije u oblik `do-while` petlje ili se koristi naredba bezuvjetnog skoka oblika:

```
void petlja()
{
    start: iskaz
        if (e) goto start;
}
```

Kod većine rekurzivnih funkcija ova se transformacije ne može provesti. Primjerice funkcija za proračun  $n$  faktoriijela nije "tail" rekurzivna, jer se u izrazu  $n * \text{factorial}(n-1)$  najprije vrši poziv funkcije `factorial`, a zatim naredba množenja.

**Funkciju `binSearch`, opisanu u prethodnom članku, može se lako transformirati u "tail" rekurzivnu funkciju:**

```
int binSearch( int a[],int x, int d, int g)
{
    int i;
    if (d > g) return -1;
    i = (d + g)/ 2;
    if (a[i] == x) return i;
    if (a[i] < x) d = i+1;
    else g = i-1
    return binSearch( a, x, d, g);
}
```

**Dalje se može provesti transformacija u iterativnu funkciju:**

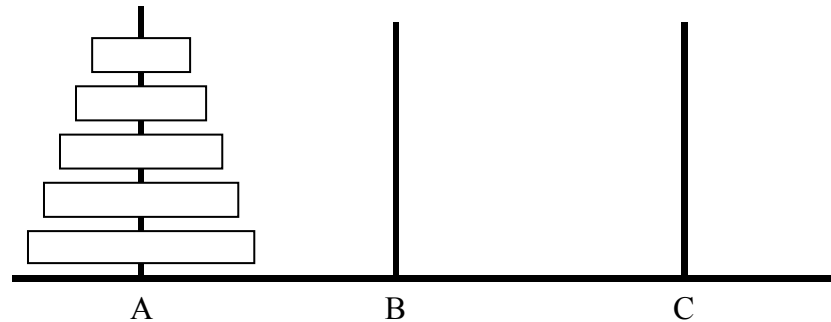
```
int binSearch( int a[],int x, int d, int g)
{start: if (d > g) return -1;
    int i = (d + g)/ 2;
    if (a[i] == x) return i;
    if (a[i] < x) d=i+1;
    else g=i-1
    goto start;
}
```

**Konačno možemo napisati iterativno tijelo funkcije u obliku while petlje:**

```
int binSearch( int a[],int x, int d, int g)
{
    while (d <= g)
    {
        int i = (d + g)/2;
        if (a[i] == x) return i;
        if (a[i] < x) d=i+1;
        else g=i-1;
    }
    return -1;
}
```

## Kule Hanoia

Obradit ćemo problem koji je, vjerojatno, najpoznatiji rekurzivni problem u kompjuterskoj literaturi. Radi se o inteligentnoj igrici koja se naziva Kule Hanoia. Problem je predstavljen na slici.



Postoje tri štapa označena s A, B i C. Na prvom štapu su nataknuti cilindrični diskovi promjenljive veličine, koji imaju rupu u sredini. Zadatak je premjestiti sve diskove s štapa A na štap B u redosljedu kako se nalaze na štapu A. Pri prebacivanju diskova treba poštovati sljedeće pravila:

1. Odjednom se smije pomicati samo jedan disk.
2. Ne smije se stavljati veći disk povrh manjeg diska.
3. Može se koristiti štap C za privremeni smještaj diskova, ali uz poštovanje prethodna dva pravila.

**U čemu je ovaj problem rekurzivan?** Pokušajmo definirati temeljni slučaj i pravilo rekurzije.

**Temeljni slučaj** - Koji je najjednostavniji slučaj kojeg svatko može riješiti. Ako kula sadrži samo jedan disk, tada je rješenje jednostavno: prebaci se taj disk na ciljni štap B.

**Rekurzivno pravilo** - Ako kula sadrži N diskova, pomicanje diskova se može izvesti u tri koraka

1. Pomakni gornji N-1 disk na pomoćni štap C.
2. Preostali donji disk sa štapa A pomakni na ciljni štap B.
3. Zatim kulu od N-1 diska s pomoćnog štapa C prebaci na ciljni štap B.

Kako napisati funkciju koji izvršava gornje pravilo. Nazvat ćemo je `pomakni_kulu`, a argumente koji će nam biti potrebni su: broj diskova koje treba pomaknuti, ime početnog štapa, ime ciljnog štapa, ime pomoćnog štapa.

```
void pomakni_kulu(int n, char A, char B, char C);
```

Također će nam biti potrebna funkcija kojom će se na prikladan način označiti prebacivanje jednog diska. Nju možemo odmah definirati sa:

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

Korištenjem ove funkcije i pravila rekurzije, funkcija `MoveTower` se može napisati na sljedeći način:

```
void pomakni_kulu(int n, char A, char B, char C)
{
    if (n == 1) // temeljni slučaj
        pomakni_disk(A, B);
    else {
        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 testiranje funkcije `pomakni_kulu()`, koristit ćemo program `hanoi.c`:

```
int main()
{
    int n = 3; /* npr. za slučaj 3 diska*/
    pomakni_kulu(n, 'A', 'B', 'C');
    return 0;
}
```

Nakon izvršenja ovog programa dobije se izvještaj o pomaku diskova oblika:

```
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
```

U ovom primjeru je očito da se pomoću rekurzije dobije fascinantno jednostavno rješenje problema, i teško da postoji neka druga metoda kojom bi se ovaj problem riješio na jednako efikasan način.

## 13.8 Složenost algoritama

Kvalitetan je onaj algoritam kojim se postiže efikasno korištenje memorijskog prostora  $M$  i prihvatljivo vrijeme obrade  $T$ .

Pri analizi složenosti algoritama uvodi se mjera "dimenzije problema", primjerice kada se obrađuju nizovi, onda dimenziju problema predstavlja duljina niza  $n$ , a kada su u pitanju kvadratne matrice, onda je dimenzija matrice istovremeno i dimenzija problema. Ako je  $n$  dimenzija problema, onda nas interesiraju funkcije  $M(n)$  i  $T(n)$ .

Veličina  $T(n)$  se često naziva vremenska složenost algoritma (engl. time complexity), dok je  $M(n)$  prostorna složenost. Veličini  $T(n)$  se pridaje znatno više značaja nego veličini  $M(n)$ , pa se pod skraćenim pojmom "složenost" (eng. complexity) obično podrazumijeva vremenska složenost.

### "Veliki-O" notacija

Analizirajmo sada segment programa u kojem treba sumirati elemente kvadratne matrice, dimenzije  $n$ . Također, označimo broj ponavljanja naredbi (pri tome se podrazumijeva da su signifikantne one naredbe u kojima se vrši zbrajanje).

NAREDBE PROGRAMA	BROJ PONAVLJANJA
-----	
S = 0 ;	.....
for (i=0; i<n; i++) {	.....n
for (j=0; j<n; j++) {	.....n*n
S = S + M[i][j]	.....n*n
-----	

Ako se uzme da prosječno vrijeme izvršenja jedne naredbe iznosi  $t_0$ , dobije se ukupno vrijeme izvršenja algoritma iznosi:

$$T(n) = (2n^2 + n) * t_0$$

Veličina  $T(n)$  je posebno važna za velike vrijednosti od  $n$ . U to slučaju, možemo zaključiti da je:

$$T(n) \leq \text{konst} * n^2$$

Ovaj se zaključak u računarskoj znanosti piše u obliku tzv. "veliki-O" notacije:

$$T(n) = O(n^2)$$

i kaže se da je  $T(n)$  "veliki  $O$  od  $n^2$ ". Funkcija  $f(n) = n^2$  predstavlja red složenosti algoritma. Uočite da vrijednost konstantnog faktora ne određuje složenost algoritma već samo faktor koji ovisi o veličini problema.

Definicija: Složenost algoritma u "veliki-O" notaciji definira se na sljedeći način:

$$T(n) = O(f(n)), \quad (\text{čita se: } T \text{ je veliki-O od } f)$$

ako postoje pozitivne konstante  $C$  i  $n_0$ , takove da je  $0 \leq T(n) \leq C * f(n)$ , za sve vrijednosti  $n \geq n_0$ .

U prethodnom je primjeru funkcija složenosti određena razmatrajući broj operacija u kojima se vrši zbrajanje. To ne treba uzeti kao pravilo, već za pojedini problem treba sagledati koje su operacije dominantne po zauzeću



procesorskog vremena. Primjerice, kod metoda pretraživanja niza to će biti naredbe usporedbe i pridjele vrijednosti.

Klasifikacija algoritama prema redu funkcije složenosti za najpoznatije klase algoritama prikazana je u sljedećoj tablici:

<i>Tip algoritma</i>	<i><math>f(n)</math></i>
Konstantan	const.
Logaritamski	$\log_2 n$
Linearan	$n$
Linearno-logaritamski	$n \log_2 n$
Kvadratni	$n^2$
Stupanjski	$n^k$ ( $k > 2$ )
Eksponencijalni	$k^n$ ( $k > 1$ )
Faktorijelni	$n!$

Ova je tablica uređena po kriteriju rastuće složenosti algoritama. Pri izradi algoritama cilj je nalaženje rješenja koje je manjeg reda složenosti, a posebno je značajan rezultat kada se nađe (najčešće približno) polinomsko ili logaritamsko rješenje nekog od eksponencijalnih problema. U nastavku bit će dan kratak opis svake od navedenih klasa algoritama.

## KONSTANTNI ALGORITMI

Konstantni algoritmi su klasa algoritama kod kojih je vrijeme rada približno konstantno i ne ovisi od veličine problema. Primjerice, program koji računa kvadratni korijen realnog broja do rezultata po pravilu dolazi poslije približno istog vremena rada bez obzira kolika je veličina broja koji je uzet za ulazni podatak.

## LOGARITAMSKI ALGORITMI

Kod logaritamskog algoritma vrijeme rada programa proporcionalno je (najčešće binarnom) logaritmu veličine problema. Imajući u vidu sporost porasta logaritamske funkcije vidi se da se ovdje radi o najefikasnijim i stoga najpopularnijim algoritmima. Tipičan predstavnik logaritamskih algoritama je binarno pretraživanje sortiranog niza prikazano u prethodnom odjeljku. Opća koncepcija logaritamskih algoritama je sljedeća:

1. Obaviti postupak kojom se veličina problema prepolovi.
2. Nastaviti razlaganje problema dok se ne dođe do veličine 1.
3. Obaviti završnu obradu s problemom jedinične veličine.

Ukupan broj razlaganja  $k$  dobije se iz uvjeta:

$$n / 2^k = 1 .$$

odnosno  $k = \log_2 n$ . Kako je po pretpostavci broj operacija koji se obavlja pri svakom razlaganju približno isti, to je i vrijeme rada približno proporcionalno broju razlaganja, odnosno binarnom logaritmu od  $n$ .

## LINEARNI ALGORITMI

Linearni algoritmi se javljaju u svim slučajevima gdje je obradom obuhvaćeno  $n$  istovjetnih podataka i gdje udvostručenje količine radnji ima za posljedicu udvostručenje vremena obrade. Opći oblik linearnog algoritma može se prikazati u vidu jedne `for` petlje:

```
for (i=0; i < n; i++)  
    {obrada koja traje vrijeme t}
```

Zanemarujući vrijeme opsluživanja `for` petlje, u ovom slučaju funkcija složenosti je  $T(n)=nt$ , pa je  $T(n) = O(n)$ .

## LINEARNO-LOGARITAMSKI ALGORITMI

Linearno-logaritamski algoritmi, složenosti  $O(n \log n)$ , spadaju u klasu veoma efikasnih algoritama jer im složenost, za veliki  $n$ , raste sporije kvadratne funkcije. Primjer za ovakav algoritam je Quicksort, koji će biti detaljno opisan kasnije. Bitna osobina ovih algoritama je sljedeća

- (1) Obaviti pojedinačnu obradu kojom se veličina problema prepolovi.
- (2) Unutar svake polovine sekvencijalno obraditi sve postojeće podatke.
- (3) Nastaviti razlaganje problema dok se ne dođe do veličine 1.

Slično kao i kod logaritamskih algoritama i ovdje je ukupan broj polovljenja  $\log_2 n$ , ali kako se pri svakom polovljenju sekvencijalno obrade svi podaci, to je ukupan broj elementarnih obrada jednak  $n \log_2 n$ , i to predstavlja rezultatni red funkcije složenosti.

### **KVADRATNI ALGORITMI**

Kvadratni algoritmi, složenosti  $O(n^2)$ , najčešće se dobijaju kada se koriste dvije `for` petlje jedna unutar druge. Primjer je dat na početku ovog poglavlja.

### **STUPANJSKI ALGORITMI**

Stupanjski algoritmi se mogu dobiti poopćenjem kvadratnih algoritama, za algoritam s  $k$  umetnutih `for` petlji složenost je  $O(n^k)$ .

### **EKSPONENCIJALNI ALGORITAMSKI**

Eksponencijalni algoritmi  $O(k^n)$  spadaju u kategoriju problema za koje se suvremena računala ne mogu koristiti, izuzev u slučajevima kada su dimenzije takvog problema veoma male. Jedan od takvih primjera je algoritam koji rekurzivno rješava igru "Kule Hanoia", opisan u prethodnom poglavlju.

### **FAKTORIJELNI ALGORITMI**

Kao primjer faktorijskih algoritama najčešće se uzima problem trgovačkog putnika. Problem je formuliran na sljedeći način: zadano je  $n+1$  točaka u prostoru i poznata je udaljenost između svake dvije točke  $j$  i  $k$ . Polazeći od jedne točke potrebno je formirati putanju kojom se obilaze sve točke i vraća opet u polaznu točku tako da je ukupni prijeđeni put minimalan.

Trivijalni algoritam za rješavanje ovog problema mogao bi se temeljiti na uspoređivanju duljina svih mogućih putanja. Broj mogućih putanja iznosi  $n!$ .

Polazeći iz početne točke postoji  $n$  putanja do  $n$  preostalih točaka. Kada odaberemo jednu od njih i dodamo u prvu točku onda nam preostaje  $n-1$  moguća putanja do druge točke,  $n-2$  putanja do treće točke, itd.,  $n+1-k$  putanja do  $k$ -te točke, i na kraju samo jedna putanja do  $n$ -te točke i natrag u polaznu točku.

Naravno da postoje efikasnije varijante algoritma za rješavanje navedenog problema, ali u opisanom slučaju, sa jednostavnim nabranjem i uspoređivanjem duljina  $n!$  različitih zatvorenih putanja, dolazimo do algoritma čije je složenost  $O[n!]$ .

## 13.9 Sortiranje

Sortiranje je postupak kojim se neka kolekcija elemenata uređuje tako da se elementi poredaju po nekom kriteriju. Kod numeričkih nizova red elemenata se obično uređuje od manjeg prema većim elementima, kod nizova stringova red se određuje prema leksikografskom rasporedu. Kod kolekcija kojoj elementi strukture imaju više članova obično se odabire jedan član strukture kao ključni element sortiranja.

Sortiranje je važno u analizi algoritama, jer su analizom različitih metoda sortiranja postavljeni neki od temeljnih kompjuterskih algoritama.

Za analizu metoda sortiranja postaviti ćemo sljedeći problem: Odredite algoritam u kojem je:

Ulaz: Niz  $A[0..n-1]$  od  $n$  elemenata.

Izlaz: Niz  $A$  sa elementima poredanim tako da je

$$A[0] \leq A[1] \leq A[2] \leq \dots \leq A[n-2] \leq A[n-1].$$

## Selekcijsko sortiranje

Započet ćemo s vjerojatno najjednostavnijom metodom koju ćemo nazvati selekcijsko sortiranje. Ideja algoritma je:

1. Pronađi najmanji element i njegov indeks  $k$ .
2. Taj element postavi na početno mjesto u nizu ( $A[0]$ ), a element koji je dotad postojao na indeksu 0 postavi na indeks  $k$ .
3. Zatim treba pronaći najmanji element počevši od indeksa 1. Kada ga pronađemo zamjenjujemo ga s elementom indeksa 1.
4. Postupak ponavljamo za sve indekse ( $2.. n-2$ ).

Primjerice za sortirati niz od 6 elemenata: 6 4 1 5 3 2 , treba izvršiti sljedeće operacije:

6	4	1	5	3	2	->	min od $A[0..5]$	zamijeni sa $A[0]$
1	4	6	5	3	2	->	min od $A[1..5]$	zamijeni sa $A[1]$
1	2	6	5	3	4	->	min od $A[2..5]$	zamijeni sa $A[2]$
1	2	3	5	6	4	->	min od $A[3..5]$	zamijeni sa $A[3]$
1	2	3	4	6	5	->	min od $A[4..5]$	zamijeni sa $A[4]$
1	2	3	4	5	6	->	niz je sortiran	

Sada možemo napisati funkciju za selekcijsko sortiranje niza  $A$  koji ima  $n$  elemenata:

```

void selectionSort(int *A, int n)
{
    int i, j;
    int imin; // indeks najmanjeg elementa u A[i..n-1]

    for (i = 0; i < n-1; i++)
    {
        // Odredi najmanji element u A[i..n-1].
        imin = i;           // pretpostavi da je to A[i]
        for (j = i+1; j < n; j++)
            if (A[j] < A[imin]) // ako je A[j] najmanji
                imin = j;      // zapamti njgov indeks

        //Sada je A[imin] najmanji element od A[i..n-1],
        // njega zamjenjujemo sa A[i].

        swap(A[i], A[imin]);
    }
}

```

### Analiza selekcijskog sortiranja

Uzet ćemo da se svaka naredba izvršava neko konstantno vrijeme. Svaka iteracija vanjske petlje (indeks  $i$ ) traje konstantno vrijeme  $t_1$  plus vrijeme izvršenja unutarnje petlje (indeks  $j$ ). Svaka iteracija u unutarnjoj petlji traje konstantno vrijeme  $t_2$ .

Broj iteracija unutarnje petlje ovisi u kojoj se iteraciji nalazi vanjska petlja:

$i$	Broj operacija u unutarnjoj petlji
0	$n-1$
1	$n-2$
2	$n-3$
$\dots$	$\dots$
$n-2$	1

**Ukupno vrijeme je:**

$$T(n) = [t_1 + (n-1) t_2] + [t_1 + (n-2) t_2] + [t_1 + (n-3) t_2] + \dots + [t_1 + (1) t_2]$$

odnosno, grupirajući članove u oblik  $t_1 ( \dots ) + ( \dots ) t_2$  dobije se

$$T(n) = (n-1) t_1 + [ (n-1) + (n-2) + (n-3) + \dots + 1 ] t_2$$

Izraz u uglatim zagradama predstavlja sumu aritmetičkog niza

$$1 + 2 + 3 + \dots + (n-1) = (n-1)n/2 = (n^2-n)/2,$$

pa je ukupno vrijeme jednako:

$$T(n) = (n-1) t_1 + [(n^2-n)/2] t_2 = -t_1 + t_1 n - t_2 n/2 + t_2 n^2/2 = O(n^2)$$

## Sortiranje umetanjem

Algoritam sortiranja umetanjem (eng. insertion sort), se temelji na postupku koji je sličan načinu kako se slažu igraće karte. Ideju demonstrirajmo nizom od  $n=6$  brojeva. Algoritam se vrši u  $n-1$  korak. U svakom koraku se umeće  $i$ -ti element u dio niza koji mu prethodi ( $A[0..i-1]$ ), tako taj niz bude sortiran.

6	4	1	5	3	2
4	6	1	5	3	2
1	4	6	5	3	2
1	4	5	6	3	2
1	3	4	5	6	2
1	2	3	4	5	6

-> ako je  $A[1] < A[0]$ , umetni  $A[1]$  u  $A[0..0]$   
-> ako je  $A[2] < A[1]$ , umetni  $A[2]$  u  $A[0..1]$   
-> ako je  $A[3] < A[2]$ , umetni  $A[3]$  u  $A[0..2]$   
-> ako je  $A[4] < A[3]$ , umetni  $A[4]$  u  $A[0..3]$   
-> ako je  $A[5] < A[4]$ , umetni  $A[5]$  u  $A[0..4]$   
-> niz je sortiran

Algoritam se može zapisati pseudokodom:

```
for (i = 1; i < n; i++)
{
    el = A[i];
    analiziraj elemente A[0 .. i-1]
    počevši od indeksa j=i-1, do indeksa j=0
        ako je el < A[j], pomakni element A[j] na mjesto A[j+1]
        inače prekini
    zatim umetni el na mjesto A[j+1]
}
```

### Analza složenosti sortiranja umetanjem

Najgori slučaj - Vanjska petla se izvršava u  $n-1$  iteracija, što daje  $O(n)$  iteracija. U unutarnjoj petlji se vrši od 0 do  $i < n$  iteracija, u najgorem slučaju vrši se također  $O(n)$  iteracija. To znači da u najgorem slučaju imamo  $O(n) \cdot O(n)$  operacija, dakle složenost je  $O(n^2)$ .

Najbolji slučaj U najboljem slučaju u unutarnjoj petlji se vrši 0 iteracija. To nastupa kada je niz sortiran. Tada je složenost  $O(n)$ . Može se uzeti da to vrijedi i kada je niz "većim dijelom sortiran" jer se tada rijetko izvršava unutarnje petlja.

Prosječni slučaj - Uzmemo li da se u prosječnom slučaju vrši  $n/2$  iteracija u unutarnjoj petlji, to također daje složenost unutarnje petlje  $O(n)$ , paje ukupna složenost:  $O(n^2)$ .



## Quicksort

Quicksort je rekurzivna metoda sortiranja kojom se u prosječnom slučaju postiže  $O(n \log_2 n)$ , što je znatno bolje od  $O(n^2)$  (koji nastaje samo u najgorem slučaju). Često se koristi u praksi. Temelji se na tehnici podijeli pa vadaj.

Neka je problem sortirati dio niza  $A[p..r]$ . Koristi se algoritam:

1. **PODIJELI.** Izaberi jedan element iz niza  $A[p..r]$  i zapamti njegovu vrijednost. Taj element ćemo nazivati pivot. Nakon toga podijeli  $A[p..r]$  u dva podniza  $A[p..q]$  i  $A[q+1..r]$  koji imaju slijedeća svojstva:
  1. Svaki element  $A[p..q]$  je manji ili jednak pivotu .
  2. Svaki element  $A[q+1..r]$  je veći ili jednak pivotu.
  3. Niti jedan podniz ne sadrži sve elemente (odnosno ne smije biti prazan).
2. **VLADAJ.** Rekurzivno sortiraj oba podniza  $A[p..q]$  i  $A[q+1..r]$ , i problem će biti riješen kada oba podniza budu imala manje od 2 elementa.

Uvjet da nijedan podniz ne bude prazan je potreban jer kada bi to bilo ispunjeno, tada bi rekurzivni problem bio isti kao originalni problem, pa bi nastala bekonačna rekurzija.

Pokažimo sada kako podijeliti podnizove da budu ispunjeni postavljeni uvjeti. Podjela se vrši funkcijom

```
int partition(int *A, int p, int r)
```

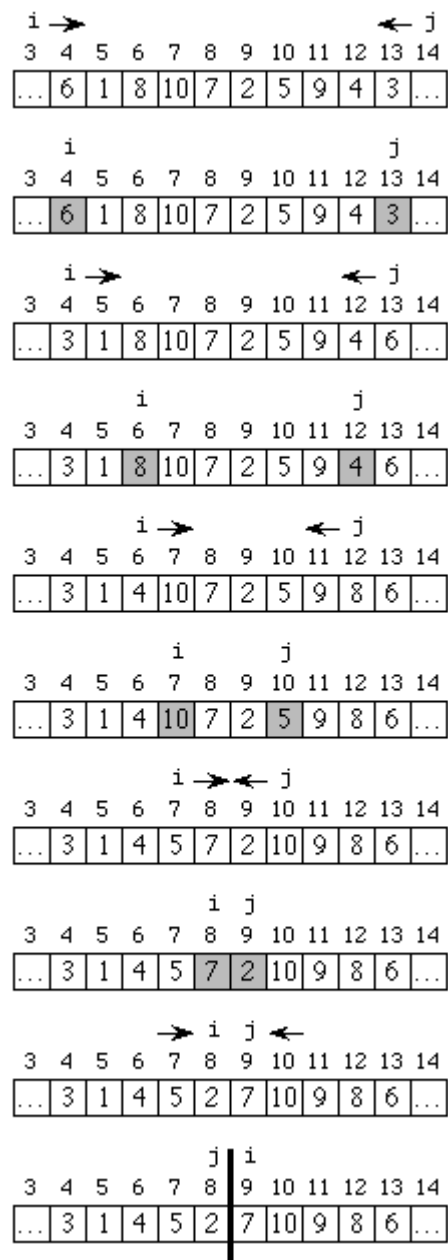
koja vraća indeks  $q$  (gdje je izvršena podjela na podnizove).

Podjela se vrši pomoću dva indeksa ( $i, j$ ) i pivota koji se odabire kao element  $A[p]$ , prema pravilu:

- $i$  pomičemo od početka prema kraju niza, dok ne nađemo element  $A[i]$  koji je veći ili jednak pivotu
- $j$  pomičemo od kraja prema početku niza, dok ne nađemo element  $A[j]$  koji je manji ili jednak pivotu
- zatim zamjenjujemo vrijednosti  $A[i]$  i  $A[j]$ , kako bi svaki svaki element  $A[p..i]$  bio manji ili jednak pivotu, a svaki element  $A[j..r]$  veći ili jednak pivotu.

- Ovaj se proces nastavlja dok se ne dobije da je  $i > j$ . Tada je podjela završena, a  $j$  označava indeks koji vraća funkcija partition. Ovaj uvjet ujedno osigurava da nijedan podniz neće biti prazan.

Ovaj postupak je ilustriran na nizu  $A[4..13]$ . Označeni su indeksi  $(i,j)$ , a potamnjeni elementi pokazuju koje elemente zamjenjujemo. Linija podjela je prikazana u slučaju kada  $i$  postane veći od  $j$ .





```

/* Podijeli A[p..r] u podnizove A[p..q] and A[q+1..r],
 * gdje je p <= q < r, i
 * svaki element A[p..q] <= od elementa A[q+1..r].
 * Početno se izabire A[p] kao pivot
 * Vraća vrijednost od q koja označava podjelu.
/*
int partition(int *A, int p, int r)
{
    int pivot = A[p]; // izbor pivot-a */
    int i = p - 1;     // index lijeve strane A[p..r]
    int j = r + 1;     // index lijeve strane A[p..r]

    while (1) {
        // Nadji sljedeći manji indeks j, A[j] <= pivot.
        while (A[--j] > pivot){}

        // Nadji sljedeći veći indeks i, A[i] >= pivot.
        while (A[++i] < pivot){}

        // Ako je i >= j, raspored je OK.
        // inače, zamijeni A[i] sa A[j] i nastavi.

        if (i < j) swap(A[i], A[j]);
        else      return j;
    }
}

```

Pomoću ove funkcije se sada iskazuje kompletni quicksort algoritam:

```

/* Sortira niz A[p..r] - quicksort algoritmom
 * Podijeli A[p..r] u podnizove A[p..q] i A[q+1..r],
 * p <= q < r, i

```

```

    * svaki element A[p..q] je <= od elementa A[q+1..r].
    * Zatim rekursivno sortiraj A[p..q] i A[q+1..r].
    /*
void quicksort(int *A, int p, int r)
{
    if (p < r) //završava kada podniz ima < od 2 elementa
    {
        int q = partition(A, p, r); /* q je pivot */
        quicksort(A, p, q); // rekursivni sort A[p..q]
        quicksort(A, q+1, r); // rekursivni sort A[q+1..r]
    }
}

```

### Analiza složenosti quicksort algoritma

Prvo uočimo da je vrijeme koje je potrebno za podjelu podnizova od  $n$  elemenata, proporcionalno broju elemenata  $cn$  (konstanta  $c > 0$ ), tj. složenost je  $O(n)$ . Analizirajmo zatim broj mogućih rekursivnih poziva.

#### Najbolji slučaj

U najboljem slučaju nizovi se dijele na dvije jednake polovine. Pošto je tada ukupan broj razlaganja jednak  $\log_2 n$  donije se da je složenost  $O(n) \cdot O(\log_2 n) = O(n \log_2 n)$ .

#### Najgori slučaj

U najgorem slučaju podjela na podnizove se vrši tako da je u jednom podnizu uvijek samo jedan element. To znači da bi tada imali  $n$  rekursivnih poziva, pa bi ukupna složenost bila  $O(n) \cdot O(n) = O(n^2)$ . Najgori slučaj nastupa kada je niz sortirani ili "skoro" sortirani.

#### Prosječni slučaj

Analiza prosječnog slučaja je dosta komplicirana. Ovdje je nećemo iznositi. Važan je zaključak da se u prosječnom slučaju dobija složenost koja je bliska najboljem slučaju  $O(n \log_2 n)$ .

### **Kako odabrati pivot element**

**U prethodnom algoritmu za pivota je odabran početni element  $A[p]$ . U praksi se pokazalo da je znatno povoljnije odabir pivota vršiti po nekom slučajnom uzorku. Recimo, ako slučajno odaberemo index  $ix$  i element  $A[ix]$  kao pivot, tada se smanjuje mogućnost da se dobije nepovoljna podjela kod skoro sortiranih nizova. Cilj je dobiti podjelu koja je bolja od omjera  $n/3:2n/3$ . Na ovaj način se dobija vjerojatnost dobre podjele koja je veća od 30%. Sljedećim postupkom se još dobija bolja vjerojatnost dobre podjele (50%). Odaberu se tri indeksa po slučajnom uzorku, a zatim se iz niza izabere jedan od ova tri elementa koji je najbliži srednjoj vrijednosti (median).**

**Dublje opravdanje ove metode ovdje neće biti dano. Pokazalo se, da uz primjenu takovog postupka, quicksort predstavlja najbrži poznati način sortiranja nizova, pa je implementiran u standardnoj biblioteci C jezika.**