UNIVERZITET METROPOLITAN BEOGRAD

CS203 - ALGORITMI I STRUKTURE PODATAKA

Analiza složenosti algoritama

Lekcija 02

PRIRUČNIK ZA STUDENTE

CS203 - ALGORITMI I STRUKTURE PODATAKA

Lekcija 02

ANALIZA SLOŽENOSTI ALGORITAMA

- Analiza složenosti algoritama
- Poglavlje 1: Složenost algoritma
- Poglavlje 2: Asimptotska složenost algoritma
- Poglavlje 3: Klase složenosti algoritama
- Poglavlje 4: Analiza vremenske složenosti algoritama
- Poglavlje 5: Vežbe
- Poglavlje 6: Zadaci za samostalan rad
- Poglavlje 7: Domaći zadatak
- Zaključak

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upoznaju sa osnovnim pojmovima u vezi analize algoritama:

- Složenost algoritma i načini merenja složenosti algoritama.
- Asimptotska složenost i tipovi notacija: Veliko O, Veliko Omega i Veliko Teta notacija.
- Analiza osnovnih numeričhih algoritama.

Jednom kada se postavi algoritam za određeni problem i ustanovi da radi tačno, sledeći korak je da se ispita količina resursa, vremena i prostora koja je potrebna da bi se algoritam izvršio. Ovaj korak se naziva analiza algoritama. Naime, algoritam koji zahteva više hiljada gigabajta RAM memorije nije uopšte upotrebljiv kod današnjih računara i pored činjenice da radi bez ijedne greške. Stoga je veoma bitno razviti što efikasniji algoritam za postavljeni problem, što je jedna od tema u okviru ove lekcije.

Poglavlje 1

Složenost algoritma

UVODNA RAZMATRANJA

Analiza algoritma predviđa performanse nekog algoritma. Najvažniji zadatak analize je da odredi vreme izvršavanja programa u funkciji ulaza, i maksimalni memorijski prostor za podatke

Složenost ili ***kompleksnost*** (complexity) je veličina memorije računara i vremena potrebnog za izvršavanje nekog programa. Postoje dve vrste složenosti algoritama, a to su prostorna i vremenska složenost.

Analiza algoritma predviđa performanse nekog algoritma. Analizom algoritama može se odrediti:

- vreme izvršavanja programa u funkciji ulaza
- ukupni ili maksimalni memorijski prostor potreban za podatke programa
- ukupna veličina programskog koda
- da li program korektno izračunava rezultat problema
- složenost programa (na primer, kako ga lako pročitati, razumeti ili promeniti program)
- koliko je program robustan (na primer, kako obrađuje neočekivane pogrešne ulaze).

Analizirajmo jedan prost i uvodni primer. Pretpostavimo da imamo dva programa koji u sebi sadrže ugneždene petlje, pa se sabiranjem broja iteracija dobija odgovarajuća jednačina koja opisuje ukupan broj iteracija petlje za izvršenja programa. Neka je broj ulaznih podataka označen sa n:

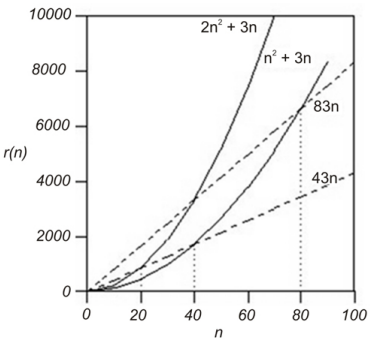
Program A

- Najgori slučaj: $2n^2+3n$ iteracija
- Najbolji slučaj: $n^2+ 3n$ iteracija.

Program B

- Najgori slučaj: 83n iteracija
- Najbolji slučaj: 43n iteracija.

Kao što možemo videti (Slika 1.1) vreme izvršavanja programa A, tj. $r(n)$ raste mnogo brže sa povećanjem broja ulaznih podataka n.



Slika 1.1 Vremena izvršavanja programa A i B za vrednost ulaznog podatka n [4]

VREMENSKA SLOŽENOST

Vremenska složenost (kompleksnost vremena, eng. time complexity) algoritma predstavlja količinu vremena koju zateva algoritam da se izvršava u funkciji veličine ulaza datog problema

Vremenska složenost algoritma meri količinu vremena koju algoritam koristi da bi se, za dati problem, izvršio u funkciji veličine ulaza.

Sistem u velikom broju slučajeva zahteva od korisnika da obezbedi gornju granicu na vreme izvršavanja programa.. Postavlja se pitanje kako da odredimo ukupno vreme za izvršenje programa.

Naravno, ukupno vreme potrebno za neki program se deli na:

- vreme prevođenja (engl. compile time) i
- vreme izvršavanja (engl. run time).

Vreme kompajliranja nije mera brzine nekog programa. Stoga je najveće briga vreme izvršavanja.

Najbolji način da se proceni vreme izvršavanja algoritma je pomoću broja operacija i broja koraka.

Operacija brojanja koraka

Operacija brojanja koraka je jedan od načina da se proceni vremenska složenost algoritma. Sastoji se iz sledećeg:

- izabrati jednu ili više operacija kao što su sabiranje, množenje i poređenje
- odrediti koliko je ovih operacija izvršeno
- identifikovati operacije koje najviše doprinose složenosti vremena.

Primer: Maksimalni element niza. Program vraća poziciju najvećeg elementa u nizu. Broj operacija koraka ćemo izračunati tako što odredimo broj poređenja elemenata niza **"*a*"** :

- kada je *n <= 0*, ne ulazi se u za petlju
- kada je *n > 0*, svaka iteracija u petlji uradi jedno poređenje

Stoga: ukupan broj poređenja je *n*.

U algoritmima sortiranja, pretraživanja i njima sličnim, npr. veličina ulaza će nama biti broj elemenata niza koji treba sortirati ili pretražiti.

PRIMERI ODREĐIVANJA VREMENSKE SLOŽENOSTI

Vremenska složenost algoritma se može proceniti pomoću broja operacija i broja koraka

Primer 1: Broj koraka u funkciji koja računa sumu niza

Neka je data sledeća funkcija:

```
public static Computable sum (Computable [] a, int n)
{
    count++; // za uslov i return
    if (a.length == 0)   return null;
    Computable sum = (Computable) a[0].zero();
    count++; // za prethodni iskaz
    for (int i = 0; i < n; i++)
    {
        count++; // za for iskaz
        sum.increment(a[i]);
        count++; // za inkrement
    }
    count++; // za poslednje izvrsavanje for iskaza
    count++; // za return
    return sum;
}
```

U prethodnoj funkciji koristimo brojač *count* kako bi odredili ukupan broj koraka. Možemo primetiti da se svaki poziv *sum* izvršava sa ukupno *2n+4* koraka kada je *0 <= n < a.length*.

Primer 2: Broj koraka u funkciji koja rekurzivno računa sumu

Neka je dat sledeći segment koda:

```
public static Computable recursiveSum(Computable [] a, int n)
{
    // Pozivač za rekurzivni metod rsum.
    count++; // za if-else iskaz
    if (a.length > 0) return rSum(a, n);
    else              return null;  // nema sta da se sabere
}
private static Computable rSum(Computable [] a, int n)
{
    if (n == 0)
    {
        count++; // za if uslov i return
        return (Computable) a[0].zero();
```

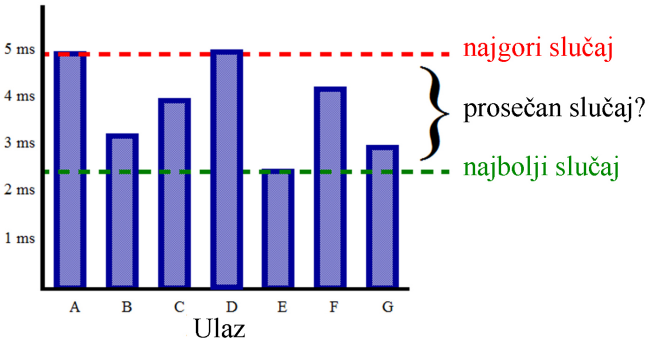
```
    }
    else
    {
        count++;  // za else blok, pozive rSum i add, i return
        return (Computable) rSum(a, n - 1).add(a[n-1]);
    }
}
```

U prethodnoj funkciji takođe koristimo brojač *count* kako bi odredili ukupan broj koraka. Možemo primetiti da je broj koraka za *recursiveSum* jednak *n+2* kada je *0 <= n < a.length*.

NAJBOLJI, NAJGORI I PROSEČAN SLUČAJ

Ponekad je veoma teško da se odredi tačno vreme izvršavanja programa. U cilju uprošćenja, u toku vremenske analize se uglavnom traže najbolji, najgori, i prosečni brojevi operacija

Ponekad je veoma teško da se odredi tačno vreme izvršavanja programa, s obzirom da program ima puno uslovnih iskaza (***if*** petlji na primer) od kojih zavisi kojim će se putem odvijati odgovarajući algoritam. Stoga, radi uprošćenja, uvedeno je da se u toku vremenske analize uglavnom traže najbolji, najgori, i prosečni brojevi operacija (engl. best, average, worst case complexities), Slika 1.2.



Slika 1.2 Najbolji, prosečni i najgori slučajevi složnosti [4]

Prosečni broj operacija je često veoma teško odrediti i definisati. Stoga uvodimo dodatno uprošćenje i ograničavamo našu analizu na najbolje i najgore brojeve operacija.

PROSTORNA SLOŽENOST ALGORITMA

Prostorna složenost algoritma je količiina memorije neophodna za potrebe izvršavanja algoritma

Prostorna složenost (engl. space complexity) označava korelaciju veličine ulaza sa brojem potrebnih memorijskih lokacija koje je potrebno alocirati za potrebe izvršavanja algoritama.

Elementi prostorne složenosti algoritma su: prostor za instrukcije, prostor za podatke, stek za izvršavanje programa. Prostor za instrukcije je prostor potreban za čuvanje prevedene verzije (engl. compiled version) instrukcija programa. Faktori od kojih zavisi složenost su: prevodilac, opcije prevođenja i ciljni računar (engl.target computer).

Prostor za podatke je prostor potreban za konstante i jednostavne promenljive kao i prostor potreban za dinamički alocirane objekte.

Prostorna složenost se nekada ignoriše jer je korišćeni prostor uglavnom minimalan i/ili očigledan, ali nekada može da postane ozbiljan problem kao i vremenska složenost.

Poglavlje 2

Asimptotska složenost algoritma

UVOD U ASIMPTOTSKU SLOŽENOST

Vremenska složenost algoritma je opisana asimptotski, odnosno, kako veličina ulaza ide do beskonačnosti, i to obično u smislu jednostavnijih funkcija

Vremenska složenost algoritma meri količinu vremena koju algoritam koristi da bi se, za dati problem, izvršio u funkciji veličine ulaza.

U slučaju kada veličina ulaza ide ka beskonačnosti kažemo da je vremenska složenost algoritma opisana asimptotski.

Funkcije koje ćemo koristiti za opis vremenske složenosti algoritma su obično proste funkcije čiji argument teži ka određenoj vrednosti ili beskonačnosti. Osnovna ideja analize funkcije je fokusiranje na stope rasta (engl. growth rates): različite funkcije sa istom stopom rasta mogu biti predstavljene korišćenjem iste klase funkcija (linearne, kvadratne, logaritamske funkcije, itd).

Neka je dat sledeći primer:

```
s = 0;
for(i = 0; i < n, i++) {  // n
    for(i = 0; i < n, i++) { // n * n
        S = S + M[i][i];  // n * n
    }
}
```

Veličina *T(n)* (vreme izvršavanja) je posebno važna za velike vrednosti ulaznog podatka *n*. U tom slučaju, možemo zaključiti da je:

$T(n) <= konst * n^2$

Funkcija $f(n)=n^2$ predstavlja red složenosti algoritma. Uočite da vrednost konstantnog faktora ne određuje složenost algoritma već samo faktor koji zavisi od veličine problema.

U prethodnom primeru funkcija složenosti je određena razmatrajući broj operacija u kojima se vrši sumiranje. To ne treba uzeti kao pravilo, već za pojedini problem treba sagledati koje su operacije dominantne po zauzimanju procesorskog vremena. Na primer, kod metoda pretraživanja niza to će biti naredbe poređenja i dodele vrednosti.

Karakteristike asimptotske složenosti

- uzimaju se samo najuticajniji članovi izraza za određeno vreme izvršavanja (umesto tačnog vremena izvršavanja, kaže se na primer ***O(n^2)***)
- opisuje ponašanje funkcije sa limitom (kada vrednost teži beskonačnosti)

TIPOVI NOTACIJA ASIMPTOTSKE SLOŽENOSTI

Najčešće korišćene notacije kada je u pitanju analiza vremenske složenosti algoritma su: Veliko O, Veliko Omega i Veliko Teta notacija

Najčešće korišćeni tipovi notacija kod asimptotske složenosti su (Slika 2.1):

a) Asimptotska gornja granica - Veliko O notacija

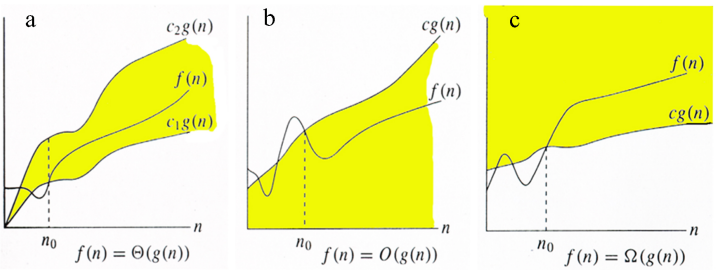
Neka je f funcija čija je vrednost realan ili kompleksan broj, a g funkcija čija je vrednost realan broj, Obe funkcije su definisane za neograničen skup realnih pozitivnih brojeva tako da je g(n) striktno pozitivno za ogromne vrednosti broja n. Kaže se:

- f(n) je O(g(n)) ako postoji pozitivna konstanta c i $n_0$tako da je f(n)≤ c * g(n) za sve n ≥ n_0

Veliko O je skup svih funkcija čiji je stepen rasta isti ili manji od g(n).
b) Asimptotska donja granica – Veliko Omega notacija

- f(n) je Ω(g(n)) ako ∃ pozitivne konstante c i n0 tako da jet 0 ≤ c*g(n)≤ f(n)∀ n ≥ n0.

Omega notacija ili asimptotski ***"veće od",*** je ustvari skup svih funkcija čiji je stepen rasta isti ili veći od *g(n). f(n)* je ograničena sa donje strane funkcijom *g(n)* u svim tačkama desno od n_0. Notacija f(n) = Ω(g(n)) se čita kao ***"f(n) je Veliko Omega od g(n)"*** , što znači da je *f(n)* asimptotski veće ili jednako *g(n)*.



Slika 2.1 Poređenje notacija: a) Teta notacija, b) Veliko O notacija, c) Omega notacija [4]

U asimptotskom smislu, *g(n)* je donji limit za *f(n)*.

c) Asimptotski tesna granica – Veliko Theta notacija

- f(n) je Θ(g(n)) ako ∃ pozitivne konstante c_1,c_2 i n_0 tako da je c_1 g(n) ≤ f(n)≤ c_2g(n)∀ n ≥ n_0

Drugim rečima: za sve n ≥ n0, funkcija f(n) je jednaka sa g(n) u granicama konstantnog faktora. Stoga kažemo:

g (n) je asimptotski tesna granica za f***(n).***

Notacija f(n) = Θ(g(n)) se čita kao ***"f(n) je Veliko Teta od g(n)"*** , što znači da je *f(n)* asimptotski jednako *g(n)* (skup funkcija čiji je stepen rasta *g(n)*).

PRIMENA VELIKO O NOTACIJE

O notacija predstavlja gornju granicu odnosno najgore vreme izvršavanja programa

Veliko O se koristi da se predstavi ***gornja granica*** vremena izvršavanja algoritma, što se drugačije zove i ***„najgori slučaj"*** .

Primer 1: Razmotrimo sledeću funkciju, $f(n)=4n^3+10n^2+5n+1$. Pod pretpostavkom da je $g(n)=n^3$, važi:

$f(n) \leqslant 5g(n)$ za sve vrednosti $n>2$

Stoga, složenost od f(n) može biti predstavljena kao O(g(n)), tj. kao $O(n^3)$.

Primer 2: Neka je data sledeća funkcija:

$f(n) = 3n+2$

Kada je c=4, 0=2, n_0 =2, tada je f(n) >= 4n

f(n) = O(n), tako da je g(n) = n

Primer 3. Neka je data sledeća funkcija:

$f(n) = 3n^2+ 4n = O(n^3)$

Za slučaj kada je c=7,n0=0 imamo

$3n^2 + 4n \leq = 3n^3+ 4n^3= 7n^3$

U cilju što efikasnijeg određivanja gornje granice algoritma pomenućemo primere nekih osnovnih pravila koja važe za Veliko O notaciju:

1. O(c f(n)) = O(f(n)), c je konstanta

$O(20\ n^3) = O(n^3)$

2. Ukoliko je O(f(n)) > O(h(n)) onda je O(f(n) + h(n)) = O(h(n)).

$O(n^2 \log n + n^3) = O(n^3)$

$O(2000n^3+ 2n! + n^{800} + 10n + 27n \log n + 5) = O(n!)$

3. O(f(n) * h(n)) = O(f(n)) * O(h(n))

$O((n^3 + 2n^2 + 3n \log n + 7)(8n^2 + 5n + 2)) = O(n^5)$

Veliko O analiza je veoma efikasan alat, ali on ima i neka ograničenja:

- Njegovo korišćenje nije pogodno za male količine ulaznih podataka.
- Za određene algoritme, konstanta koja je dobijena na osnovu Veliko O pristupa može biti previše velika da bi bila praktična. Na primer, ako je vreme izvršavanja nekog algoritma određeno formulom ***2N log N*** , a nekog drugog jednačinom *1000N*, onda je više verovatno da je prvi algoritam bolji, iako je njegova stopa rasta veća.

PRIMENA VELIKO OMEGA I VELIKO THETA NOTACIJE

Veliko Omega notacija definiše najbolje vreme izvršavanja algoritma. Notacija f(n) = Ω(g(n)) se čita kao "f(n) je omega od g(n)", što znači da je f(n) asimptotski veće ili jednako g(n)

Veliko Omega se koristi da se predstavi donja granica vremena izvršavanja algoritma, što se često zove i ***„najbolji slučaj"*** algoritma.

Primer 1: Razmotrimo funkciju $f(n)=4n^3+10n^2+5n+1$. Pod pretpostavkom da je $g(n)=n^3$, i da važi $f(n)\geqslant 4.g(n)$ za sve vrednosti $n>0$.

Stoga, složenost funkcije f(n) može biti predstavljena kao Ω(g(n)) , tj kao. $\Omega(n^3)$.

Primer 2:

$10n+7 = \Omega(n)$, 10n+7 je asimptotski jednako n

$100_n^3 - 3 = \Omega(n^3)$

$3n^3+2n+6 = \Omega(n)$

$8n^4 + 9n^2 \neq \Omega(n^3)$

Veliko Theta se koristi da se prestavi ***prosečan slučaj*** algoritma.
Primer 3: Razmotrimo funkciju, $f(n)=4n^3+10n^2+5n+1$
Pod pretpostavkom da je $g(n)=n^3$, $4.g(n) \leqslant f(n) \leqslant 5.g(n)$ za sve velike vrednosti broja n.

Stoga, složenost funkcije f(n) može biti predstavljena kao θ(g(n)), tj. kao $\theta(n^3)$.

Primer 4:

- $3n^2 + 4n = \Theta(n^2)$
- Ako je $c_1=3$, $c_2=7$ i $n_0 = 0$
- $3n^2 \leq 3n^2 + 4n \leq 7n^2$ za sve n,$n \geq n_0$
- f(n) je ograničena na gore in a dole nekom funkcijom g(n) za sve tačke desno od n_0.

Primer 5:

- $f(n)= 3n^2 + 2n + 1$, $f(n) = \Theta(n^2)$
- kada je $c_1 = 3$, $c_2 = 4$, $n_0 =1 + \sqrt{2}$
- $3n^2 \leq 3n^2 + 2n + 1 \leq 4n^2 (n > 0)$

- → $n \geq 1 + \sqrt{2}$ tako da je $n_0 = 1 + \sqrt{2}$

Primer 6:

- $10n + 7 = \Theta(n)$, $10n + 7$ je asimptotski jednako n
- $100n^3 - 3 = \Theta(n^3)$;
- $12n + 6 = \Theta(n)$; $3n^3 + 2n + 6 \neq \Theta(n)$; $8n^4 + 9n^2 \neq \Theta(n)$;

PRIMERI: ODREĐIVANJE SLOŽENOSTI ALGORITAMA

U velikom broju primera određivanja vremenske složenosti, koriste se uobičajena matematička sumiranja, čije su formule navedene u nastavku

Navodimo formule za sume nekih nizova, koje se trivijalno mogu izračunati uz pomoć matematičke indukcije, a nama će koristiti prilikom ocenjivanja složenosti. Sledeća matematička sumiranja se obično koriste u analizi algoritama (Slika 2.2) .

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

$$1 + 2 + 3 + \ldots + (n-1) + n = \frac{n(n+1)}{2} = O(n^2)$$

$$a^0 + a^1 + a^2 + a^3 + \ldots + a^{(n-1)} + a^n = \frac{a^{n+1} - 1}{a - 1} = O(a^n)$$

$$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{(n-1)} + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1 = O(2^n)$$

Slika 2.2 Matematička sumiranja koja se koriste kod Veliko O notacije [1]

Primer 1. Petlje

```
// O(n)
for(int i = 0; i < n; i++)
    sum  = sum - i;

//O(n^2)
for(int i = 0; i < n*n; i++)
    sum  = sum + i;

// O(log n)
i = 1;
while (i<=n){
    sum = sum + i;
    i = i * 2;
}
```

Primer 2. Ugneždene petlje

```
// Primer 1 - O(n^2)
sum  = 0;
```

```
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        sum += i*j;
// Primer 2 - O(nlogn)
i = 1;
while (i<=n){
    j = 1;
    while (i<=n){
        // iskazi konstantne slozenosti
        j *=2;
    }
    i += 1;
}
```

Primer 3. Sekvence izjava

```
for(int j = 0; j < n * n; j++)
    sum  = sum + j;
for(int k = 0; k < n; k++)
    sum  = sum + k;
System.out.print("sum is now" + sum);
```

Složenost je $O(n^2)+O(1)=O(n^2)$.

Poglavlje 3

Klase složenosti algoritama

OSNOVNI TIPOVI ALGORITAMA

Osnovni tipovi algoritama su konstantan, logaritamski, linearan, linearno-logaritamski, kvadratni, stepeni, eksponencijalni i faktorijelni

Klasifikacija algoritama prema redu funkcije složenosti za najpoznatije klase algoritama prikazana je na sledećoj slici (Slika 3.1):

Tip algoritma	*f(n)*
Konstantan	const.
Logaritamski	$\log_2 n$
Linearan	n
Linearno-logaritamski	$n\log_2 n$
Kvadratni	n^2
Stepeni	n^k (k>2)
Eksponencijalni	k^n (k>1)
Faktorijelni	n!

Slika 3.1 Klasifikacija algoritama [2]

Pri izradi algoritama cilj je nalaženje rešenja koje je manjeg reda složenosti, a posebno je značajan rezultat kada se nađe (najčešće približno) polinomsko ili logaritamsko rešenje nekog od eksponencijalnih problema.

Konstantni algoritmi

Kod konstantnih algoritama vreme rada je približno konstantno i ne zavisi od veličine problema. Primera radi, program koji računa kvadratni koren realnog broja do rezultata, po pravilu, dolazi posle približno istog vremena rada, bez obzira kolika je veličina broja koji je uzet za ulazni podatak.

Logaritamski algoritmi

Kod logaritamskog algoritma vreme rada programa proporcionalno je (najčešće binarnom) logaritmu veličine problema. Imajući u vidu sporost porasta logaritamske funkcije vidi se da se ovde radi o najefikasnijim i stoga najpopularnijim algoritmima. Tipičan predstavnik logaritamskih algoritama je *binarno pretraživanje* sortiranog niza prikazano u prethodnoj lekciji. Opšta koncepcija logaritamskih algoritama je sledeća:

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 uslova $n/2^k=1$, odnosno $k=log_2 n$. Kako je po pretpostavci broj operacija koji se obavlja pri svakom razlaganju približno isti, to je i vreme rada približno proporcionalno broju razlaganja, odnosno binarnom logaritmu od n.

OPIS OSTALIH ALGORITAMA

Linearni algoritmi rastu srazmerno veličini ulaznih podataka. Linearno logaritamski spadaju u klasu veoma efikasnih algoritama. Kvadratni algoritmi se sastoje iz dve ugneždene petlje

Linearni algoritmi. Linearni algoritmi se javljaju u svim slučajevima, gde je obradom obuhvaćeno n istovetnih podataka, i gde udvostručenje količine radnji ima za posledicu udvostručenje vremena obrade. Opšti oblik linearnog algoritma može se prikazati u vidu jedne ***for*** petlje:

for (i=0; i > n; i++) {obrada koja traje vreme t}

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

Linearno-logaritamski algoritmi. Linearno-logaritamski algoritmi, složenosti O(n logn), spadaju u klasu veoma efikasnih algoritama, jer im složenost za veliko n, raste sporije od kvadratne funkcije. Primer za ovakav algoritam je ***Quicksort*** . Bitna osobina ovih algoritama je sledeća:

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

Kvadratni algoritmi. Kvadratni algoritmi, složenosti $O(n^2)$, najčešće se dobijaju kada se koriste dve ***for*** petlje jedna unutar druge.

Stepeni algoritmi. Stepeni algoritmi se mogu dobiti uprošćenjem kvadratnih algoritama. Za algoritam sa k umetnutih ***for*** petlji složenost je $O(nk)$.

Eksponencijalni algoritmi. Jedan od primera je algoritam koji rekurzivno rešava igru ***Kule Hanoja*** , koga ćemo analizirati u nastavku lekcije.

Faktorijelni algoritmi. Kao primer faktorijelnih algoritama najčešće se uzima ***Problem trgovačkog putnika*** . Problem je formulisan na sledeći način: neka je zadato $n+1$ tačaka u prostoru i neka je poznata udaljenost između svake dve tačke j i k. Polazeći od jedne tačke potrebno je formirati putanju kojom se obilaze sve tačke i vraća opet u polaznu tačku, tako da je ukupni pređeni put minimalan. Naravno, postoje efikasnije varijante algoritma za rešavanje navedenog problema, ali u slučaju sa jednostavnim nabrajanjem i upoređivanjem daljina $n!$ različitih zatvorenih putanja, dolazimo do algoritma čije je složenost $O(n!)$ putanja. Broj mogućih putanja iznosi $n!$

REKURENTNE RELACIJE

Rekurencije su prirodan način da se predstavi vreme izvršavanja podeli i osvoji algoritma

Rekurentne relacije su jedan koristan alat za analizu složenosti algoritama. Rekurentne relacije idu ruku pod ruku sa strategijom "podeli i osvoji", jer nam daju prirodan način da predstavimo vreme izvršavanja "podeli i osvoji" algoritama. Rekurentna relacija je jednačina ili nejednačina koja opisuje vrednost funkcije u odnosu na male vrednosti ulaznih veličina. Na primer, jedna uobičajena rekurentna relacija je:

$$T(n) = \begin{cases} O(n) & if\ n = 1 \\ 2T(n/2) + O(n) & if\ n > 1 \end{cases}$$

čije rešenje je složenosti $T(n) = O(n\ lg\ n)$.

Rekurentne relacije mogu da imaju puno oblika. Na primer, rekurzivni algoritam može da se podeli na podprobleme različitih veličina, kao što je podela na 2/3 i 1/3. Ako koraci podele i spajanja zahtevaju linearno vreme, takav algoritam će se svesti na rekurentnu relaciju oblika $T(n) = T(2n/3) + T(n/3) + \backslash O(n)$.

Potproblemi neće nužno biti konstantna frakcija veličine originalnog problema. Na primer, rekurzivna verzija linearnog pretraživanja će kreirati samo jedan potproblem koji će imati samo jedan element manje u odnosu na originalni problem. Svaki rekurzivni poziv će zahtevati konstantno vreme plus vreme za pozive koje on napravi, što će da se svede na rekurentnu relaciju oblika $T(n) = T(n-1) + \backslash O(n)$

Rekurentne relacije mogu biti određene korišćenjem Master teoreme, i o tome ćemo pričati u lekciji o rekurziji.

Kao što je pokazano u prethodnim primerima, složenost algoritama binarnog pretraživanja, sortiranja selekcijom je T(n) = T(n/2) + c odnosno
T(n) = T(n-1) + O(n) . U narednoj tabeli su sumirane osnovne rekurentne relacije.

Rekurentna relacija	*Rezultat*	*Primer*
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binarna pretraga, Euklidov NZD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linearno pretraživanje
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	Najveći element niza
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Kule Hanoja
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Rekurzivni Fibonači algoritam

Slika 3.2 Rekurentne relacije [1]

POREĐENJE OSNOVNIH FUNKCIJA RASTA

Poređenjem osnovnih funkcija rasta stičemo dobar uvid u složenost različitih vrsta algoritama. Na osnovu toga možemo uvek proceniti da li je neki algoritam dovoljno brz ili ne

U narednoj tabeli (Slika 3.3) su date neke od osnovnih funkcija rasta kao i prikaz kako se menja veličina izlaza kada se veličina ulaznih podataka udvostruči sa *n = 25* na *n = 50*.

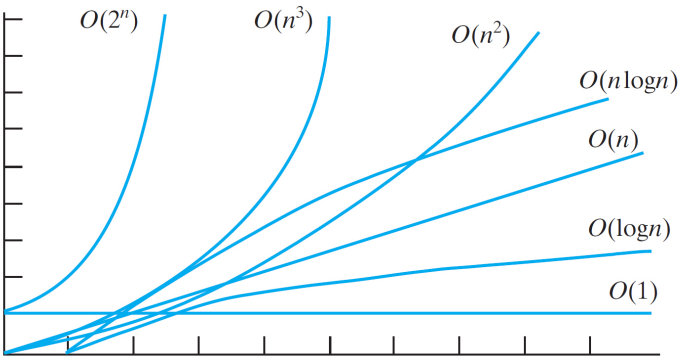
Slika 3.3 prikazuje neke korišćene funkcije rasta i promene funkcija rasta za *n* od *n = 25* do *n = 50*.

Funkcija	*Naziv*	$n = 25$	$n = 50$	$f(50)/f(25)$
$O(1)$	Konstantna	1	1	1
$O(\log n)$	Logaritamska	4.64	5.64	1.21
$O(n)$	Linearna	25	50	2
$O(n \log n)$	Linearno-logaritamska	116	282	2.43
$O(n^2)$	Kvadratna	625	2,500	4
$O(n^3)$	Kubna	15,625	125,000	8
$O(2^n)$	Eksponencijalna	3.36×10^7	1.27×10^{15}	3.35×10^7

Slika 3.3 Promena stope rasta raznih vrsta algoritama [1]

Funkcije su prema vremenu izvršavanja uređene na sledeći način, kako je i ilustrovano (Slika 3.4).

$O(1)>O(\log n)> O(n)> O(n\log n)>O(n^2)>O(n^3)>O(2^n)$



Slika 3.4 Funkcija raste kako se n povećava [1]

Poglavlje 4

Analiza vremenske složenosti algoritama

ANALIZA ALGORITMA ZA ODREĐIVANJE NZD DVA BROJA

Algoritam iscrpne pretrage proverava da li je broj k (k = 2, 3, 4, itd) zajednički delilac brojeva n1 i n2, i to proverava sve dok k ne postane veće od n1 ili n2. Njegova složenost je O(n)

U prethodnoj lekciji o osnovnim algoritamskim strategijama, opisali smo algoritam iscrpne pretrage (brute-force) koji pronalazi NDZ dva cela broja *min*.

Algoritam iscrpne pretrage proverava da li je broj *k* (gde je *k = 2, 3, 4,* itd) zajednički delilac brojeva *n1* i *n2*, i to proverava sve dok *k* ne postane veće od *n1* ili *n2*. Pogledajmo kod:

```
public static int gcd(int m, int n) {
    int gcd = 1;
    for (int k = 2; k <= m && k <= n; k++) {
        if (m % k == 0 && n % k == 0)
            gcd = k;
    }
    return gcd;
}
```

U slučaju da je m ≥ n složenost ovog algoritma je očigledano *O(n)*.

Nekada je umesto pretrage mogućeg delioca u petlji od *1* do *n*, pogodnije i efikasnije pretraživati u petlji od *n* do *1*. Jednom kada naiđemo na zajedničkog delioca, to je onaj koji je ustvari i najveći, pa tu možemo da prekinemo pretragu. Delilac broja *n* ne može biti veći od *n / 2*, ali sa druge strane moguće je da je upravo *n* NZD za *n* i *m*, pa ne bi bilo tačno samo pustiti da petlja pretrage ide od *1* do *n/2*. Stoga, uz uključen ovaj slučaj, korektan algoritam bi imao sledeći oblik:

```
public static int gcd(int m, int n) {
    int gcd = 1;

    if (m % n == 0) return n;

    for (int k = n / 2; k >= 1; k--) {
        if (m % k == 0 && n % k == 0) {
            gcd = k;
```

```
            break;
        }
    }

    return gcd;
}
```

Uz pretpostavku da je m ≥ n, ***for*** petlja će se izvršiti maksimalno *n/2* puta, što u najgorem slučaju skraćuje broj operacija na pola u odnosu na prethodni algoritam. Složenost algoritma je i dalje *O(n)*, ali praktično, ovo je duplo brži algoritam od prethodnog.

Veliko O notacija daje veoma dobru teorijsku procenu složenosti i efikasnosti algoritma. Međutim, dva algoritma iste vremenske složenosti ne moraju biti uvek isto efikasni. Kao što smo primetili iz prethodnih primera, imamo dva algoritma iste vremenske složenosti, ali je poslednji algoritam očigledno bolji i brži.

ANALIZA ALGORITAMA ZA FIBONAČIJEVE BROJEVE

Rekurzivno rešenje Fibonačijevih brojeva je algoritam kvadratne složenosti. Problem kod ovog rešenja je taj što se neke funkcije za iste argumente pozivaju više puta

Već ste imali prilike da vidite program koji primenom rekurzije vrši određivanje Fibonačijevih brojeva:

```
public static long fib(long index) {
    if (index == 0 || index == 1) // osnovni slucaj
        return index;
    else // Redukcija i rekurzivni pozivi
        return fib(index - 1) + fib(index - 2);
}
```

Dokažimo da je složenost ovog algoritma $O(2^n)$. Uzećemo da je indeks člana koji određujemo *n*. Označimo sa *T(n)* složenost algoritma koji pronalazi *fib(n)* , a sa *c* označimo konstantno vreme koje je potrebno za poređenje indeksa sa vrednostima 0 i 1. Očigledno, *T(1)* i *T(0)* su jednaki *c*. Stoga, u opštem slučaju imamo (Slika 4.1):

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &\leq 2T(n-1) + c \\ &\leq 2(2T(n-2) + c) + c \\ &= 2^2T(n-2) + 2c + c \end{aligned}$$

Slika 4.1 Vremenska složenost algoritma Fibonačijevih brojeva [1]

Ovde možemo pokazati da je složenost algoritma $T(n) = O(2^n)$.

Moguće je poboljšati rekurzivan algoritam da bi izbegli dupliranje poziva metode *fib* sa istim argumentom, što smo već obradili u okviru L01 i priče o ***dinamičkom programiranju*** . Mi, naravno, znamo da se novi Fibonačijev broj formira sabiranjem prethodna dva Fibonačijeva broja. Tako, ukoliko koristimo dve promenljive *f0* i *f1* da sačuvamo prethodne Fibonačijeve brojeve, onda novi Fibonačijev broj *f2*, može biti direktno određen sabiranjem brojeva *f0* i *f1*.

```
public static long fib(long n) {
    long f0 = 0; // Za fib(0)
    long f1 = 1; // Za fib(1)
    long f2 = 1; // Za fib(2)

    if (n == 0)      return f0;
    else if (n == 1) return f1;
    else if (n == 2) return f2;

    for (int i = 3; i <= n; i++) {
        f0 = f1;
        f1 = f2;
        f2 = f0 + f1;
    }
    return f2;
}
```

Očigledno, složenost iterativnog algoritma je $O(n)$. Ovo je veoma veliko poboljšanje u odnosu na eksponencijalni rekurzivni $O(2^n)$.

Poglavlje 5

Vežbe

ANALIZA ALGORITAMA - PRIMERI 1-2 (20 MIN.)

Cilj narednih primera je da se izvrši vremenska analiza i odredi vreme izvršavanja korišćenjem asismptotske notacije i brojanja koraka

Napomena: Za neke korake u algoritmu je potrebno konstantno vreme, na primer za čitanje vrednosti elementa na određenoj poziciji u nizu, za inicijalizaciju neke promenljive, za ispis vrednosti promenjive. Takve korake, sa konstantnim vremenom, ćemo ocenjivati sa O(1).

Primer 1: Oceniti vremensku složenost sledećeg algoritma (10 min):

```
for (int i = 1; i <= n; i++) {
    k = k + 5;
}
```

Za izvršavanje naredbe koja uvećava vrednost promenljive *k* :

k = k + **5**;

potrebno nam je konstantno vreme, koje ćemo označiti konstantom *c* . ***for*** petlju određuje brojač *i*, koji se inicijalno postavlja na vrednost 1 i petlja se izvršava dokle god je vrednost brojača *i* manja ili jednaka od već definisane promenljive n.

Kolika god da je vrednost promenljive n, ***for*** petlja će se izvršavati tačno *n* puta. Složenost izvršavanja ***for*** petlje je stoga:

$T(n)$ = (konstanta c)*n = $O(n)$.

Primer 2: Oceniti vremensku složenost sledećeg algoritma (10 min):

```
for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= n; j++) {
  k = k + i + j;
 }
}
```

Za naredbu dodele:

k = k + i + j;

je potrebno konstantno vreme *c*.

Za jednu iteraciju spoljašnje petlje, odredjenu iteratorom *i*, izvršiće se čitava unutrašnja petlja određena iteratorom *j*. Spoljašnja ***for*** petlja se izvršava tačno *n* puta. Slično spoljašnjoj petlji, unutrašnja petlja se takođe izvršava *n* puta. Dakle, vremenska složenost algoritma je:

$T(n)$ = (a constant c)$*n*n$ = $O(n^2)$

ANALIZA ALGORITAMA - PRIMERI 3 -4 (15 MIN)

Cilj sledećih primera je da se izvrši vremenska analiza i odredi vreme izvršavanja korišćenjem asismptotske notacije i brojanja koraka

Primer 3: Oceniti vremensku složenost algoritma (10 min):

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        k = k + i + j;
    }
}
```

U primeru imamo ugneždenu petlju. Spoljašnja petlja se izvršava tačno n puta. Broj iteracija unutrašnje petlje je određen trenutnim rednim brojem iteracije spoljašnje petlje.

Kada je i=1, u spoljašnjoj petlji, unutrašnja petlja se izvršava jednom.

Kada je i=2, u spoljašnjoj petlji, unutrašnja petlja se izvršava dva puta.

Kada je i=3, u spoljašnjoj petlji, unutrašnja petlja se izvršava tri puta. I tako redom.

Naredba dodele:

k = k + i + j;

kao i u prethodnim primerima ima konstantnu vremensku složenost. Složenost ovog primera je:

T(n) = c + 2c + 3c + 4c + . . . + nc = c n(n + 1)/2 = O(n ^{2})

Primer 4: Oceniti vremensku složenost sledećeg algoritma (5 min):

```
for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= 20; j++) {
  k = k + i + j;
 }
}
```

Ponovo imamo primer sa ugneždenim petljama. Spoljašnja petlja je odredjena iteratorom i koji uzima, redom vrednosti od 1 do n. Spoljašnja petlja se dakle, izvršava n puta.

Unutrašnja petlja se uvek izvršava tačno 20 puta, jer je odredjena iteratorom j koji dobija vrednosti od 1 do 20, redom. Unutrašnja petlja ne zavisi od broja n. Naredba dodele:

k = k + i + j;

ima konstantnu složenost, c. Dakle, vremenska složenost ovog algoritma zavisi samo od broja n koji određuje broj iteracija spoljašnje petlje.

Vremenska složenost je u ovom slučaju linearna i računa se na sledeći način:

$T(n) = 20*c*n = O(n)$

ANALIZA ALGORITAMA - PRIMER 5 (10 MIN)

Cilj ovog primera je da se izvrši vremenska analiza i odredi vreme izvršavanja korišćenjem asismptotske notacije i brojanja koraka

Oceniti vremensku složenost sledećeg algoritma (10 min):

```
for (int j = 1; j <= n; j++) {
    n = n / 2;
}
```

Ovaj primer je nešto drugačiji od prethodnih. Ne možemo olako odrediti broj iteracija for petlje, jer se broj n menja pri svakoj njenoj iteraciji. Naredba:

n = n / 2;

ima konstantnu vremensku složenost. Dakle, u toku jedne iteracije for petlje vrednost iteratora i se uvećala za jedan i tako „približila“ vrednosti n, ali vrednost n nije konstantna. U svakoj iteraciji vrednost n petlje se smanjuje, biva duplo manja.

Razmotrimo slučaj kada je n=16.

- j=1 n=16
- j=2 n=8
- j=3 n=4
- j=4 n=2

uslov j <= n; više nije ispunjen

Razmotrimo još jedan slučaj, kada je n=128.

- j=1 n=128
- j=2 n=64
- j=3 n=32
- j=4 n=16
- j=5 n=8
- j=6 n=4

uslov j <= n; više nije ispunjen

Primećujemo da se veličina n brze smanjije, nego što se vrednost iteratora j uvećava. U ovom slučaju složenost algoritma je logaritamska, i to logaritam n za osnovu 2($\log_2 n$). Medjutim, mozemo usvojiti da je vremenska slozenost algoritma:

T(n) = O(log n)

Poglavlje 6

Zadaci za samostalan rad

ZADACI IZ ANALIZE ALGORITAMA (20 MIN)

Na osnovu materijala sa predavanja i vežbi uraditi samostalno sledeće primere

Primer 1 : Izbrojati broj iteracija u sledećim petljama (5 min):

```
int count = 1;
while (count < 30) {
  count = count * 2;
}
```
(a)

```
int count = 15;
while (count < 30) {
  count = count * 3;
}
```
(b)

```
int count = 1;
while (count < n) {
  count = count * 2;
}
```
(c)

```
int count = 15;
while (count < n) {
  count = count * 3;
}
```
(d)

Slika 6.1 Primer 1 [1]

Primer 2 : Koliko zvezdica će biti prikazano ako je n jednako 10? (5 min)

```
for (int i = 0; i < n; i++) {
  System.out.print('*');
}
```
(a)

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    System.out.print('*');
  }
}
```
(b)

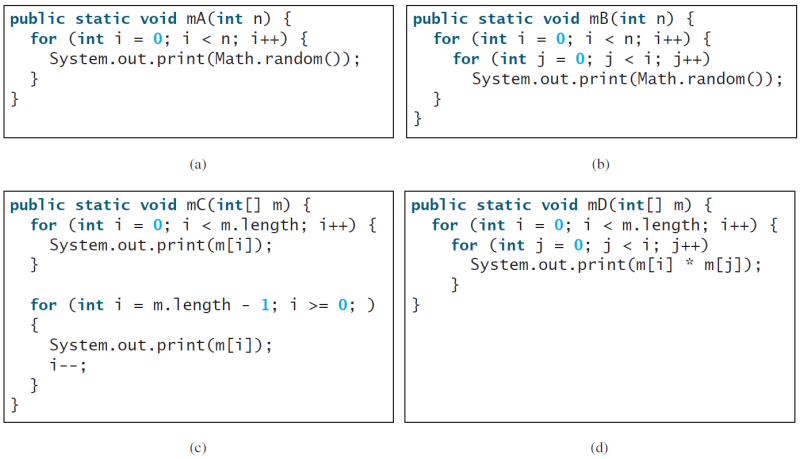
```
for (int k = 0; k < n; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      System.out.print('*');
    }
  }
}
```
(c)

```
for (int k = 0; k < 10; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      System.out.print('*');
    }
  }
}
```
(d)

Slika 6.2 Primer 2 [1]

Šta će se desiti ako je n jednako 20? Koristiti Veliko O notaciju da ocenite vremensku složenost algoritma.

Primer 3 : Upotrebiti Veliko O notacija za procenu vremenske složenosti sledećih metoda (5 min):



Slika 6.3 Primer 3 [1]

Primer 4: Analizirati sledeći algoritam sortiranja (5 min):

```
for (int i = 0; i < list.length - 1; i++) {
 if (list[i] > list[i + 1]) {
  swap list[i] with list[i + 1];
  i = -1;
 }
}
```

DODATNI ZADACI ZA SAMOSTALNI RAD (70 MIN)

Na osnovu materijala sa predavanja i vežbi uraditi samostalno i sledeće dodatne primere

Primer 5: Za svaki od sledećih isečaka programa (20 min):

- Dati vremensku analizu izvršavanja algoritma korišćenjem Veliko O notacije.
- Implementirati kod i pokrenuti program za različite vrednosti *N*.
- Uporediti vašu analizu sa stvarnim vremenima izvršavanja.

```
// Fragment 1
for( int i = 0; i < n; i++ )
 sum++;
// Fragment 2
for( int i = 0; i < n; i += 2 )
 sum++;
// Fragment 3
for( int i = 0; i < n; i++ )
 for( int j = 0; j < n; j++ )
  sum++;
// Fragment 4
for( int i = 0; i < n; i++ )
 sum++;
```

```
for( int j = 0; j < n; j++ )
 sum++;
// Fragment 5
for( int i = 0; i < n; i++ )
 for( int j = 0; j < n * n; j++ )
  sum++;
// Fragment 6
for( int i = 0; i < n; i++ )
 for( int j = 0; j < i; j++ )
  sum++;
// Fragment 7
for( int i = 0; i < n; i++ )
 for( int j = 0; j < n * n; j++ )
  for( int k = 0; k < j; k++ )
   sum++;
// Fragment 8
for( int i = 1; i < n; i = i * 2 )
 sum++;
```

Povremeno, povećanje veličine ugneždenih petlji može dati precenjenu vrednost vremena izvršavanja Veliko O notacije. Ovaj slučaj se dešava kada se petlja koja je u samoj unutrašnjosti retko izvršava. Uraditi istu analizu kao u prethodnom primeru za sledeći isečak koda:

```
for( int i = 1; i <= n; i++ )
 for( int j = 1; j <= i * i; j++ )
  if( j % i == 0 )
   for( int k = 0; k < j; k++ )
    sum++;
```

Primer 6: Većinski element niza *A* veličine *N* je onaj element koji se pojavljuje više od *N* / 2 puta (stoga, može da postoji samo jedan takav element). Na primer, niz {3, 3, 4, 2, 4, 4, 2, 4, 4} ima većinski element (4), dok niz {3, 3, 4, 2, 4, 4, 2, 4} nema. Kreirati algoritam koji pronalazi većinski element ako postoji, a izveštava korisnika u slučaju da ne postoji. Koje je vreme izvršavanja vašeg algoritma? (*Nagoveštaj*: Postoji rešenje vremenske složenosti *O*(*N*)). (20 min)

Zadatak 1. (*Podniz istih brojeva* - 30 min) Napisati program vremenske složenosti *O*(*n*) koji od korisnika zahteva da unese niz celih brojeva, gde je poslednji član 0, i pronalazi najduži podniz istih brojeva. U nastavku je primer izvršavanja programa:

```
Unesi seriju brojeva koja se zavrsava brojem 0:
  2 4 4 8 8 8 8 2 4 4 0
Najduza sekvenca istih brojeva pocinje od indeksa 3 i ima 4 vrednosti broja 8
```

Poglavlje 7

Domaći zadatak

DOMAĆI ZADATAK - PRAVILA

Posebno obratiti pažnju na sledeća pravila u vezi izrade domaćih zadataka

Svaki student dobija od asistenta sopstvenu kombinaciju domaćeg zadatka.

Online studenti bi trebalo mailom da se najave, kada budu želeli da krenu sa radom na predmetu i prikupljanjem predispitnih obaveza.

Odgovarajući NetBeans (Eclipse ili Visual Studio) projekat koji predstavlja rešenje domaćeg zadatka smestiti u folder CS203-DZ02-Ime-Prezime-BrojIndeksa. Zipovani folder CS203-DZ02-Ime-Prezime-BrojIndeksa poslati predmetnom asistentu (lazar.mrkela@metropolitan.ac.rs) u mejlu sa naslovom (subject) CS203-DZ02, inače se neće računati.

Studenti iz Niša predispitne obaveze predaju asistentu u Nišu (jovana.jovanovic@metropolitan.ac.rs i uros.lazarevic@metropolitan.ac.rs).

Student tradicionalne nastave ima 7 dana, od dana kada je dobio mail sa domaćim zadatkom, da uradi i pošalje rešenje za maksimalan broj poena.

Ukoliko student pošalje domaći nakon tog roka, najviše može da ostvari 50% od maksimalnog broja poena.

Studenti online nastave imaju rok da predaju rešene domaće zadatke 10 dana pre termina ispita u ispitnom roku u kome polažu CS203 Algoritmi i strukture podataka.

Vreme za izradu: 2h.

Zaključak

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće:

Analiza algoritma predviđa performanse nekog algoritma. Najvažniji zadatak analize je da odredi vreme izvršavanja programa u funkciji ulaza, kao i maksimalni memorijski prostor za podatke. Najbolji način da se odredi vreme izvršavanja algoritma je da se izvrši brojanje operacija u okviru neke od petlji kreiranog algoritma.

Složenost (ili kompleksnost) **algoritma** je veličina memorije računara i vremena potrebnog za izvršavanje nekog programa. Vremenska složenost (kompleksnost vremena, eng. time complexity) algoritma meri količinu vremena koju zahteva algoritam da se izvršava u funkciji veličine ulaza datog problema.

Vremenska složenost algoritma je opisana asimptotski, odnosno, kako veličina ulaza ide do beskonačnosti, i to obično u smislu jednostavnijih funkcija. Postoje tri tipa notacija kada je u pitanju analiza vremenske složenosti i to: Veliko O, Veliko Omega i Veliko Teta notacija.

Veliko O notacija predstavlja gornju granicu odnosno najgore vreme izvršavanja programa. Kod primene Veliko O notacije mogu se ignorisati multiplikativne konstante i ne-dominatni članovi u funkciji. Izborom najdominantnijeg člana u funkciji rasta na pravi način određujemo Veliko O. Veliko Omega notacija definiše najbolje vreme izvršavanja nekog algoritma. Veliko Teta notacija daje opseg prosečnih slučajeva izvršavanja algoritma.

Osnovni tipovi algoritama, prema asimptotskoj notaciji, su konstantan, logaritamski, linearan, linearno-logaritamski, kvadratni, stepeni, eksponencijalni i faktorijelni (od najboljeg do najgoreg).

Linearni algoritmi rastu srazmerno veličini ulaznih podataka. Linearno logaritamski spadaju u klasu veoma efikasnih algoritama. Kvadratni algoritmi se sastoje iz dve ugneždene petlje. Stepeni algoritmi se dobijaju uprošćenjem kvadradnih algoritama. Primer eksponencijalnog algoritma su Hanojske kule, dok je primer faktorijalnog algorima problem trgovačkog putnika.

Poređenjem osnovnih funkcija rasta stičemo dobar uvid u složenost različitih vrsta algoritama. Na osnovu toga možemo uvek proceniti da li je neki algoritam dovoljno brz ili ne.

REFERENCE

Korišćena literatura

[1] D. Liang, Introduction to Java Programming, Comprehensive Version, 10th edition, Prentice Hall, 2014

[2] M.A. Weiss, Data Structures and Problem Solving Using Java, 3rd edition, Addison Wesley, 2005.

[4] R. Sedgewick, K.Wayne, Algorithms, 4th edition, Pearson Education, 2011.

[5] T H. Cormen, C E. Leiserson, R L. Rivest, C. Stein, Introduction to Algorithms, 3th edition, The MIT Press, 2009.