Programiranje I

Složenost algoritama

Složenost algoritama

- Nijesu svi algoritmi koji rješavaju neki problem jednako kvalitetni.
- Jedna od mjera kvaliteta algoritama je složenost.
- Tipovi složenosti su:
  - Vremenska složenost
    - Koje vrijeme je potrebno za izvršavanje algoritma.
    - Proporcionalno je broju operacija (aritmetičkih, logičkih ili nekih drugih) koje algoritam izvršava.
    - Dakle, procjena vremenske složenosti se svodi na brojanje operacija u algoritmu.

Složenost algoritama

- Prostorna složenost
  - Koliko memorije zauzimaju podaci korišćeni u algoritmu.
  - Određuje se sabiranjem memorijskog prostora svih promjenljivih korišćenih u algoritmu.
- Komunikaciona složenost
  - Određuje koliko je potrebno komunikacije procesora sa periferijama, diskovima i memorijom prilikom izvršavanja programa.
  - Posebno je značajna kod paralelnih izvršavanja programa kada razni procesori međusobno komuniciraju.

Vremenska složenost

- Mi ćemo se detaljnije zadržati na vremenskoj složenosti.
- Vremenska složenost kaže da je vrijeme izvršavanja algoritma proporcionalno broju operacija u algoritmu:

  vrijeme = broj operacija × neki konstantni interval

- Generalno, ovo ne važi za malu količinu ulaznih podataka, već samo za veliki obim podataka.
- Teško je odrediti svaku operaciju u algoritmu, a teško je generalno i porediti operacije kao što su sinus ugla, množenje i npr. operacije poređenja.

Vremenska složenost: Sekvence i selekcije

- Kod sekvence se broje operacije koje se izvrše. Na primjer:

```
a = x + y + z;
b = a + 5;
c = a * b;
```

→ Složenost je 3 sabiranja i 1 množenje.

- Kod selekcije se uzima varijanta sa više operacija:

```
if(a > 3)
      c = a + b + 3;
else
      c = a  + 1;
```

→ 1 poređenje i 2 sabiranja (uzeta "gora" varijanta).

- Ovdje nismo računali operacije dodjele vrijednosti =.

Vremenska složenost: Ciklusi

- Kod ciklusa se broj operacija tokom jednog izvršavanja ciklusa množi sa brojem ponavljanja ciklusa:

```
i = 1;
while(i <= N) {
      a = a + b;
      i = i + 1;
}
```

→ 2N sabiranja i N+1 poređenje (N ponavljanja po 2 sabiranja i N+1 provjera i<=N)

```
for(i=0; i<N; i+=2) {
      a = a * b + i;
}
```

→ N sabiranja, N/2+1 poređenje i N/2 množenja

Vremenska složenost: Ugnježđeni ciklusi

- Ugnježđeni ciklusi:

```
for(i=1; i<=N; i++) {
    for(j=1; j<=N; j++) {
        ***K operacija***
    }
}
```

Složenost je N^2K operacija (unutrašnja petlja), N^2+N inkrementiranja promjenljivih i i j, i N^2+2N+1 poređenje

- Sa svakom novom ugnježđenom petljom, složenost se uvećava onoliko puta koliko ima iteracija te petlje. Stoga se izbjegavaju rješenja sa velikim brojem ugnježđenih petlji.

Vremenska složenost: Ulazni podaci

- Kod algoritama koji sadrže kombinacije ciklusa i uslova, ponekad je teško procijeniti složenost.
- Sami ulazni podaci mogu značajno uticati na složenost. Na primjer, sortiranje relativno sortiranog niza zahtijeva manje operacija od niza sa nasumično raspoređenim elementima ili niza koji je sortiran u suprotni poredak.
- Ukoliko znamo nešto o ulaznim podacima, može se vršiti sofisticiranija analiza.

Najgora i prosječna složenost

- Šta raditi kada ne postoji jedinstvena veza ulaznih podataka i složenosti? Postoje dvije strategije:
  - Analiza najgoreg slučaja (worst case analiza)
    - Uzme se najgori mogući slučaj i kaže se da algoritam ima toliku vremensku složenost u najgorem slučaju (ovo je češća strategija).
  - Analiza prosječnog slučaja (average case analiza)
    - Za svaku klasu ulaznih podataka, potrebno je znati broj operacija i vjerovatnoću pojave svake klase. Prosječan broj operacija se dobija:

Vjerovatnoća ulaznog podatka date klase

$$\sum_{i=1}^{K} p_i N_i = p_1 N_1 + p_2 N_2 + \ldots + p_K N_K$$

Broj operacija za datu klasu ulaznih podataka

Prosječna složenost

- Za vjerovatnoće važi:

$$\sum_{i=1}^{K} p_i = p_1 + p_2 + \ldots + p_K = 1$$

- **Primjer:** Parni i neparni brojevi (to su klase ulaznih podataka) se pojavljuju sa jednakom vjerovatnoćom. Obim operacija kod parnih brojeva je reda veličine **N**, kod neparnih **2N**. Šta je najgori, a šta prosječni slučaj?
- **Odgovor:** Najgori slučaj je **2N**, dok je prosječni

$$\frac{1}{2}N + \frac{1}{2}2N = \frac{3}{2}N$$

Rast složenosti. Veliko O

- Rast složenosti algoritma (vremenska, prostorna) sa povećanjem veličine ulaza n je korisna mjera za poređenje algoritama.
- Rast funkcija se obično opisuje koristeći notaciju **veliko O** (eng. big-O notation).
- **Definicija:** Neka su **f(x)** i **g(x)** funkcije koje preslikavaju skup **Z** ili **R** u skup **R**. Za funkciju f(x) se kaže da je **O(g(x))** ako postoje konstante **C** i $\mathbf{x_0}$ takve da važi:

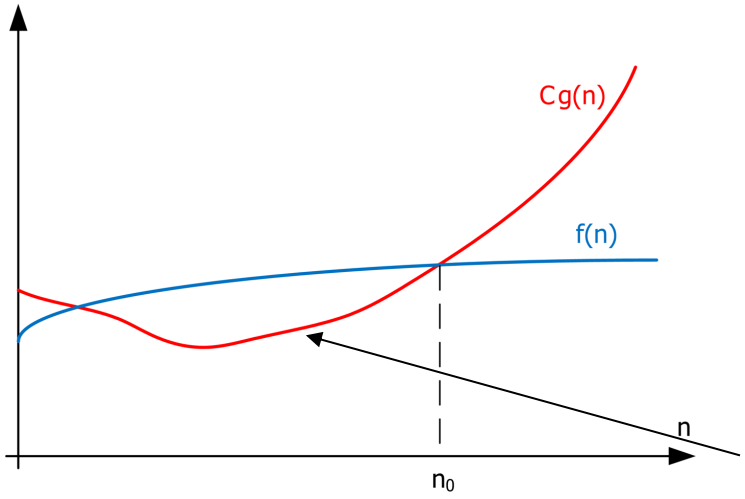
$$|f(x)| \leq C|g(x)| \quad \text{za svako } x > x_0.$$

Veliko O

- U analizi složenosti algoritama, f(x) i g(x) su uvijek pozitivne funkcije. Stoga se definicija velikog O svodi na (umjesto x koristićemo n, kao veličinu ulaznih podataka):

  f(n) ≤ Cg(n) za svako $n > n_0$

- Ako želimo da pokažemo da je f(n) reda O(g(n)), dovoljno je da pronađemo jedan par (C, n_0).



Motivacija uvođenja veliko O notacije je uspostavljanje gornje granice rasta funkcije f(n) za veliko n. Ova granica je određena funkcijom g(n) koja je obično dosta jednostavnija od f(n).

Interesuje nas samo veliko n, tj. u redu je da bude **f(n) > Cg(n)** za $n \leq n_0$.

Veliko O: Primjeri

- Pokazati da je $f(n) = n^2+2n+1$ reda kompleksnosti $O(n^2)$.
  Dokaz: Za $n>1$ važi $n^2+2n+1 \le n^2+2n^2+n^2 \Rightarrow n^2+2n+1 \le 4n^2$
  Prema tome, za $C=4$ i $n_0=1$ važi:
  $f(n) \le Cn^2$ za svako $n>n_0 \Rightarrow$ **$f(n)$ je $O(n^2)$**

- Pokazati da je $f(n) = 3n^4-8n^3+7$ reda kompleksnosti $O(n^4)$.
  Dokaz: Za $n>1$ važi:
  $|3n^4-8n^3+7| \le 3n^4+|8n^3|+7n^4 \le 3n^4+8n^4+7n^4 = 18n^4$
  Prema tome, za $C=18$ i $n_0=1$ važi:
  $f(n) \le Cn^4$ za svako $n>n_0 \Rightarrow$ **$f(n)$ je $O(n^4)$**

- Pokazati da je $f(n) = 22$ reda kompleksnosti $O(1)$.

Korisna pravila za Veliko O

- Ako je k konstanta, k≠0, onda $O(kg(n)) = O(g(n))$.
- Ako $f_1(n)$ ima složenost $O(g_1(n))$, a $f_2(n)$ $O(g_2(n))$, onda $f_1(n)+f_2(n)$ ima složenost $O(max(g_1(n), g_2(n)))$. Negdje se u literaturi može naći $O(g_1(n)+g_2(n))$, ali je ona veća od $O(max(g_1(n), g_2(n)))$.
- Ako $f_1(n)$ ima složenost $O(g_1(n))$, a $f_2(n)$ složenost $O(g_2(n))$, onda $f_1(n)*f_2(n)$ ima složenost $O(g_1(n)*g_2(n))$.
- Za bilo koji polinom $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$, gdje su a_0, a_1, …, a_n realni brojevi, f(n) ima složenost $O(n^k)$.
- Generalno, u zbiru proizvoljnog broja funkcija, najbrža određuje složenost zbira. Na primjer, složenost funkcije

  $f(n) = 12\log(n) + \log^3(n) + 4n\log(n) + 33n^2 + n^3/10$ je **$O(n^3)$**.

Veliko O: Česte kompleksnosti

- Ako je f(n) reda $O(n^2)$, da li je takođe reda $O(n^3)$?
  Da! n^3 raste brže od n^2, pa n^3 raste brže i od f(n).
- Cilj je pronaći najmanju jednostavnu funkciju g(n) za koju je f(n) reda O(g(n))!
- Funkcije g(n) koje se često sreću (poređane po kompleksnosti):
  - 1 – Konstanta
  - log(n) – Logaritamska
  - $\sqrt{n}$ – Korijena
  - n – Linearna
  - n log(n) – Linearitmična (eng. *linearithmic*)
  - n^2 – Kvadratna
  - n^3 – Kubna
  - 2^n – Eksponencijalna
  - n! – Faktorijel

Veliko O: Neki algoritmi

Operacija	Složenost
Binarna pretraga niza	$O(\log_2 n)$
Provjera je li broj prost	$O(\sqrt{n})$
Brute force pretraga niza	$O(n)$
Merge sort	$O(n \log_2 n)$
Ponovljeni minimum	$O(n^2)$
Bubble sort	$O(n^2)$
Proizvod matrica	$O(n^3)$
Ugnježdene for petlje (k nivoa)	$O(n^k)$
Generisanje svih podskupova datog skupa	$O(2^n)$
Generisanje svih permutacija datog skupa	$O(n!)$

Primjer 1: Nedostajući broj

- Niz A sadrži N različitih prirodnih brojeva iz opsega [1, 2, ..., N+1], što znači da jedan broj iz opsega nedostaje. Cilj je pronaći nedostajući prirodan broj.
- Napisati funkciju koja za argument ima niz A i njegovu dužinu N, i koja vraća nedostajući broj iz opsega [1, 2, ..., N+1].
- Npr. ako je niz A = [3,4,1,5], funkcija treba da vrati broj 2.
- Vremenska složenost funkcije treba da bude O(N).

```
int nedostajuci(int A[], int N) {
    int suma = 0, i;
    for(int i = 0; i < N; i++)
        suma += A[i];
    return (N+1)*(N+2)/2 - suma;
}
```

Jedan prolazak kroz niz od N elemenata ima složenost O(N)

Primjer 2: Je li broj prost?

- Napisati funkciju koja za argument ima prirodan broj N i koja vraća 1 ako je broj prost i 0 u suprotnom.
- Koja najmanja složenost se može postići? Odgovor: $O(\sqrt{n})$
- Direktno rješenje podrazumijeva provjeru djeljivosti broja N sa svim prirodnim brojevima od 2 do N-1, ili (malo bolje rješenje) od 2 do N/2.
- Ako je M djelilac broja N, onda je i N/M djelilac broja N (simetrični djelioci).
- U paru simetričnih djelioca, jedan mora biti manji od ili jednak $\sqrt{n}$.

```
int jeLiProst(int N) {
    int i;
    for(i = 2; i * i <= N; i++)
        if(N % i == 0)
            return 0;
    return 1;
}
```

Primjer 3: Ravnoteža niza

- Imamo niz A od N cijelih brojeva. Posmatrajmo cijeli broj P, takav da važi 0<P<N, koji predstavlja tačku presjeka niza na dva podniza: A[0], A[1], ..., A[P−1] i A[P], A[P+1], ..., A[N−1].
- Napisati funkciju koja za argument ima niz A i njegovu dužinu N, i koja vraća minimalnu apsolutnu razliku između ova dva podniza za svako moguću vrijednost P.
- Na primjer, ako je niz A = [3,1,2,4,3], ovaj se niz može podeliti na 4 različita načina, i vrijednosti apsolutnih razlika su:
  - P = 1, razlika |3−10| = 7
  - P = 2, razlika |4−9| = 5
  - P = 3, razlika |6−7| = 1
  - P = 4, razlika |10−3| = 7

  Dakle, funkcija treba da vrati broj 1 za ovaj niz A.
- Vremenska složenost funkcije treba da bude O(N).

Primjer 4: Najčešći element niza

- Elementi niza cijelih brojeva su prirodni brojevi manji od 1000. Funkcija treba da vrati element niza koji se najčešće pojavljuje.

```
int najcesciElement(int x[], int N) {
    int i, vrijednosti[1000] = {0};
    for(i = 0; i < N; i++)
        vrijednosti[x[i]]++;
    int najcesci = vrijednosti[0];
    for(i = 1; i < 1000; i++)
        if(najcesci < vrijednosti[i])
            najcesci = vrijednosti[i];
    return najcesci;
}
```

Jedan prolazak kroz niz od N elemenata i jedan prolazak kroz niz od 1000 elemenata.

- Niz vrijednosti broji pojave elemenata niza x, i to:
  - vrijednosti[0] broji elemente sa vrijednošću 0
  - vrijednosti[1] broji ... 1
  - vrijednosti[2] broji ... 2
  - vrijednosti[999] broji ... 999.
- U naredbi vrijednosti[x[i]]++, vrijednost elementa x[i] služi kao indeks!

Brojanje elemenata niza: Primjene

- Određivanje broja različitih elemenata u nizu.
- Određivanje nedostajućih elemenata u određenom opsegu vrijednosti.
- Provjera da li je niz validna permutacija (postoje svi elementi iz opsega [1,N] i pojavljuju se tačno jedanput).
- Sortiranje niza cijelih brojeva sa složenošću O(N)!
  Uslov: Elementi niza su ograničeni po vrijednosti, npr. u opsegu [-1 000 000, 1 000 000].

Sortiranje nizova kao međukorak

- Određene zadatke je pogodnije riješiti ukoliko je niz sortiran, tj. ukoliko je najmanja složenost koja se može postići bez sortiranja veća od $O(n \log_2 n)$, treba razmotriti sortiranje niza kao međukorak.
- Tipičan primjer je traženje medijane niza. Medijana niza je element niza za koji važi da postoji jednak broj manjih i većih elemenata od njega. Primjer: Medijana niza [8,3,9,7,6,1,3] je broj 6, jer postoje tri manja (1, 3 i 3) i tri veća elementa (7, 8 i 9).
- U slučaju niza sa parnim brojem elemanata, medijana je jednaka aritmetičkoj sredini dva središnja elementa sortiranog niza.
- Najmanje složen algoritam određivanja medijane niza je:
  - sortiraj niz (složenost $O(n \log_2 n)$)
  - uzmi središnji element (dužina niza neparna) ili arit. sred. dva središnja elementa (dužina niza parna).

qsort funkcija

- Najjednostavniji način sortiranja niza u C-u je koristeći ugrađenu funkciju **qsort**. Objasnimo njenu upotrebu na primjeru.

```
#include <stdio.h>
#include <stdlib.h>

int porediElemente(const void *, const void *);

int main () {
  int i, niz[] = {41, 17, 99, 81, 20, 35};
  qsort(niz, 6, sizeof(int), porediElemente);
  for (i=0; i<6; i++)
     printf("%d ", niz[i]);
}

int porediElemente(const void *a, const void *b) {
  return *(int*)a - *(int*)b;
}
```

Posljednji parametar funkcije qsort je pokazivač na funkciju koja poredi dva elementa niza. Funkcija treba da vrati:

- broj **< 0** ako element na koji pokazuje a treba da dođe ispred elementa na koji pokazuje b;
- broj **> 0** ako element na koji pokazuje a treba da dođe iza elementa na koji pokazuje b;
- **0** ako su elementi jednaki.

Sortiranje nizova: Primjene

- Određivanje broja različitih elemenata u nizu (sortirati niz i brojati pojave x[i] != x[i+1]).
- Određivanje nedostajućih elemenata u određenom opsegu (sortirati niz i detektovati pojave x[i]+1 != x[i+1]).
- Određivanje maksimalnog proizvoda dva ili više elemenata niza (sortirati niz i razmatrati ekstreme).
- Određivanje dominantnog elementa niza (onaj koji se pojavljuje više od N/2 puta, gdje je N dužina niza).

Primjer 5: Podniz sa najvećom sumom

- Dat je niz prirodnih brojeva A dužine N. Odrediti podniz (sekvenca uzastopnih elemenata) koji ima najveću sumu.
- Dakle, treba naći uređeni par (i,j), gdje je $0 \leq i \leq j \leq N-1$, za koje je suma $A[i]+A[i+1]+\cdots+A[j-1]+A[j]$ maksimalna.
- Ilustrovaćemo 3 rješenja ovog problema, čije su složenosti $O(N^3)$, $O(N^2)$ i $O(N)$.

Podniz sa najvećom sumom: Rješenje 1

```
maksSuma = A[0];
for(i=0; i<N; i++) {
    for(j=i; j<N; j++) {
        suma = 0;
        for(k=i; k<=j; k++)
            suma += A[k];
        if(suma > maksSuma)
            maksSuma = suma;
    }
}
```

Direktno rješenje. Određujemo sumu elemenata između svakog i i j, za koje važe $0 \le i \le j \le N-1$, i pronalazimo maksimum.

Pošto imamo tri petlje, složenost rješenja je $O(N^3)$.

Podniz sa najvećom sumom: Rješenje 2

```
maksSuma = A[0];
for(i=0; i<N; i++) {
    suma = 0;
    for(j=i; j<N; j++) {
        suma += A[j];
        if(suma > maksSuma)
            maksSuma = suma;
    }
}
```

Za fiksirano i, određujemo sumu elemenata svakog podniza koji počinje indeksom i, i pronalazimo maksimum.

Pošto imamo dvije petlje, složenost je $O(N^2)$.

Podniz sa najvećom sumom: Rješenje 3

```
suma = 0;
maksSuma = A[0];
for(i=0; i<N; i++) {
    suma += A[i];
    if(suma > maksSuma)
        maksSuma = suma;
    if(suma < 0)
        suma = 0;
}
```

Sumiramo redom elemente sve dok je suma>0. Ažuriramo maksSuma po potrebi. Kad suma postane negativna, vraćamo je na 0 i krećemo nanovo da sumiramo. Protumačiti!

Pošto imamo jednu petlju, složenost je O(N).