STRUKTURE PODATAKA I ALGORITMI

Predavanje 02

Ponavljanje

- Pravila koja će nam pomoći:
  - Ako nam treba novi tip podataka
    - Trebaju li nam operacije (metode)? Ako da: **klasa**; ako ne: **struktura**
    - Kreirati dvije nove datoteke; u .h obavezno staviti *include guard*
  - Svi članovi koji nisu metode moraju biti privatni
    - Metode mogu biti javne ili privatne
  - Za podešavanje početnog stanja objekta možemo koristiti konstruktor
    - Problem: ako radimo polje objekata, onda mora postojati konstruktor bez parametara
  - Kreirajte dodatne metode prema želji i potrebi
    - Za inicijalizaciju, dohvaćanje vrijednosti, postavljanje vrijednosti, ...
  - `stringstream`: za izradu stringa od više dijelova te za pretvaranje stringa u neki primitivni tip

BRZINA RADA INFORMACIJSKOG SUSTAVA

Algebra visoka škola za primijenjeno računarstvo

Uvod

- Osnovni gradivni elementi **informacijskog sustava** su **programi** (tj. **aplikacije**) i **baze podataka**
- Što napraviti kad korisnik sustava nije zadovoljan brzinom rada sustava?
  - To je tema ovog predavanja, ali ne na razini cijelog sustava već na razini pojedinačnih programa
- Prilikom izrade i u fazi održavanja važno je uočiti dijelove sustava koji ne rade efikasno
  - Optimizaciju (poboljšavanje) treba raditi na pravim mjestima
  - Optimizacija na krivom mjestu neće donijeti očekivane rezultate

Algebra visoka škola za primijenjeno računarstvo

Primjeri i diskusija

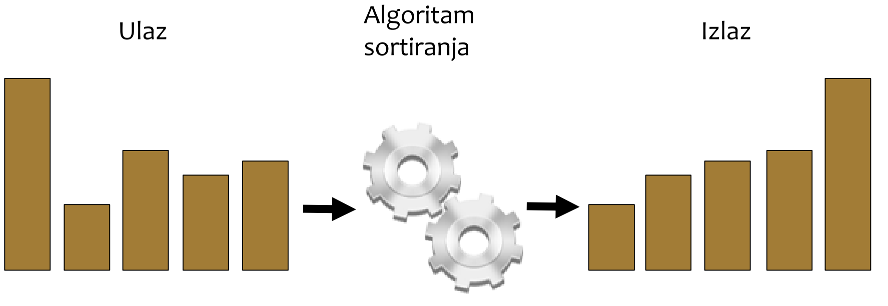
- Situacija 1: što je problem u informacijskom sustavu u primjeru 1 i kako ga popraviti?
- Situacija 2: što je problem u informacijskom sustavu u primjeru 2 i kako ga popraviti?

ANALIZA SLOŽENOSTI ALGORITAMA

Algebra
visoka škola za
primijenjeno računarstvo

Uvod

- Algoritam: dobro definiran postupak koja uzima ulaze i pretvara ih u izlaze
- Primjerice:



Dobar program je brz program

- Smatrat ćemo da je program **bolji** što je **brži** u izvođenju
  - Ubrzavanjem programa ga ujedno i poboljšavamo
  - Količinu potrošnje memorije i ostalih resursa nećemo razmatrati
- **Brzinu programa** određuje:
  1. Odabir programskog jezika
  2. Kvaliteta programiranja (tj. implementacije)
  3. Hardver i operacijski sustav
  4. **Složenost korištenih algoritama**

Složenost korištenih algoritama

- Na ovom predavanju nas zanima jedino utjecaj složenosti korištenih algoritama
  - Primjerice, u nekom sortiranju nabavka jačeg računala može popraviti brzinu izvođenja za 5%, a promjena algoritma za 300%
- Brži (dakle, bolji) program je onaj koji koristi **jednostavnije algoritme**
- **Složenost algoritma** ovisi o sljedećim elementima:
  - Koraci algoritma
  - Količina ulaznih podataka

Koraci u rješavanju problema

- Izrada programskog rješenja nekog problema se obično sastoji od barem sljedećih koraka:
  1. Izrada algoritma uz analizu i uvažavanje ***a priori*** provjere složenosti algoritma
  2. Implementacija algoritma u nekom programskom jeziku (izrada **programa**)
  3. ***A posteriori*** provjera složenosti programa
- *A priori* analiza – analiza trajanja izvođenja algoritma kao posljedica količine ulaznih podataka (tj. predviđanje)
- *A posteriori* analiza – analiza izvođenja programa na nekom stvarnom računalu (tj. mjerenje)

Tipovi analize algoritama (1/5)

- **Vrijeme izvođenja** (engl. *running time*) je vrijeme potrebno da izvođenje algoritma dođe do kraja
  - Označavat ćemo ga sa $\boldsymbol{T(n)}$, gdje je n broj ulaznih podataka
- Tipovi analiza:
  - Najgori slučaj (engl. *worst case scenario*)
  - Najbolji slučaj (engl. *best case scenario*)
  - Srednji slučaj (engl. *average case scenario*)

Algebra visoka škola za primijenjeno računarstvo

Tipovi analize algoritama (2/5)

- **Najgori slučaj** daje gornju granicu vremena izvođenja
  - n je broj ulaznih podataka koji su poredani na najgori mogući način
    - Primjerice, ako želimo pronaći broj 49 u polju od 5 elemenata, najgori slučaj bi bilo polje { 1, 14, 4, 5, 49 }
  - $T(n)$ je vrijeme izvođenja algoritma u najgorem slučaju
    - U našem primjeru $T(5) = 5$, što znači da se mora napraviti 5 logičkih operacija da bi algoritam došao do kraja
      - Općenito, za polje od n elemenata je $T(n) = n$
- **Daje apsolutnu garanciju da algoritam neće raditi dulje od $T(n)$, bez obzira na broj i vrstu ulaznih podataka**

Algebra visoka škola za primijenjeno računarstvo

Tipovi analize algoritama (3/5)

- **Najbolji slučaj** daje donju granicu vremena izvođenja
  - Gledamo određeni tip ulaza za koji algoritam radi najbrže
  - Primjerice, potraga za elementom 49 traje 1 logičku operaciju ako je ulaz polje { 49, 1, 14, 4, 5 }
    - U najboljem slučaju je T(n) = 1 bez obzira na veličinu polja

Tipovi analize algoritama (4/5)

- **Srednji slučaj** daje prosječno vrijeme izvođenja
  - Teško za izračunati jer se mora izračunati prosjek izvršavanja za sve moguće uniformno distribuirane ulazne podatke
  - Primjerice, intuitivno znamo da ćemo element 49 u nekom ulaznom polju u prosjeku pronaći na sredini polja
    - U srednjem slučaju će T(n) = n / 2

Algebra
visoka škola za primijenjeno računarstvo

Tipovi analize algoritama (5/5)

- Najbolji slučaj nam ne govori puno
- Izračun prosječnog slučaja je dosta kompleksan, u nekim slučajevima i nemoguć
- U praksi je najzanimljiviji **najgori slučaj** iz razloga:
  - Svako izvođenje sigurno neće dulje trajati od najgoreg slučaja (znamo gornju granicu)
  - Najlakši je za izračunati

NOTACIJA VELIKO O

Uvod

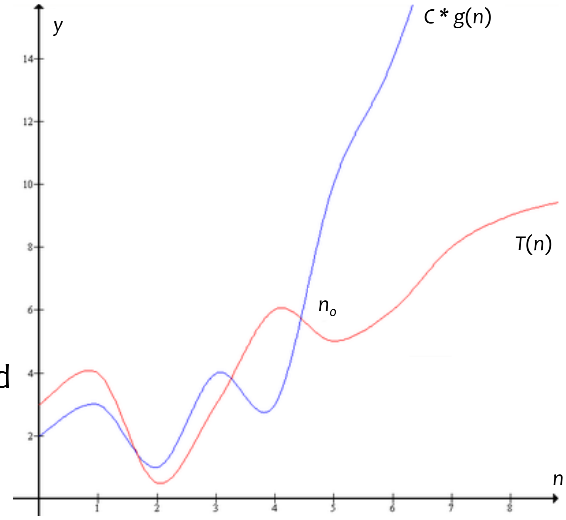
- Promatrat ćemo skupinu notacija poznatu pod nazivom Bachmann-Landau notacije ili **asimptotske notacije**
  - Veliko *O*,
  - Veliko Ω (omega),
  - Veliko Θ (theta), ...
- U analizi algoritama, notacije služe za **klasificiranje algoritama u klase** prema tome kako se ponašaju s porastom broja ulaznih podataka
- Omogućavaju nam da izrazimo kojom brzinom raste vrijeme izvođenja kad broj ulaznih podataka raste
  - To izražavamo usporedbom s nekim standardnim funkcijama

Formalna definicija (pojednostavljena)

- Uglavnom nećemo davati matematičke definicije
  - "Informatičari ih i tako preskaču, a matematičari će uvijek naći zamjerku" (Tkalac)
- Danas ćemo napraviti iznimku od pravila pa ćemo dati pojednostavljenu definiciju notacije veliko *O*
- Neka su $T(n)$ i $g(n)$ funkcije gdje je n pozitivni realni broj
- Pišemo $T(n) = O(g(n))$ akko postoje dva broja C i n_0 takvi da vrijedi $T(n) <= C*g(n)$ za svaki $n > n_0$
  - Kažemo da je $T(n)$ ograničena s $g(n)$ (tj. da $g(n)$ ograničava $T(n)$)
    - To znači da je, počevši od nekog broja, $g(n)$ uvijek veća od $T(n)$

Primjer ograničavanja

- Kažemo da funkcija g(n) ograničava funkciju T(n)
  - Postoje C (npr. C = 1) i n_0 (npr. n_0 = 5) takvi da je $T(n) <= C*g(n)$ za svaki $n > n_0$
- Možemo reći i ovako: "za sve brojeve veće od n_0 će T(n) biti manji ili jednak $C*g(n)$"



Izvor: Wikipedia

Posljedice ograničenja

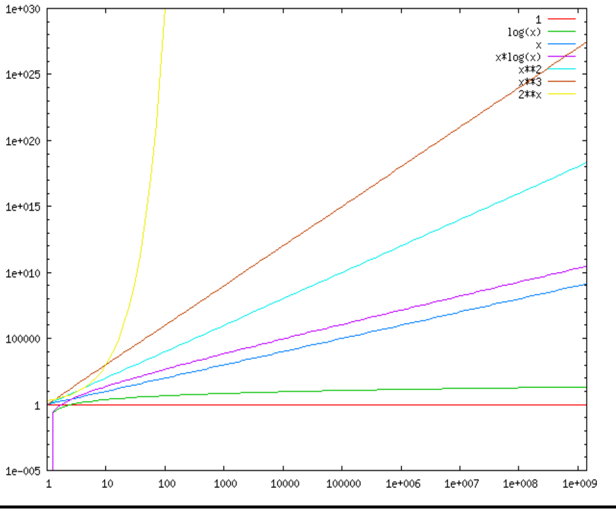
- Funkcija T(n) nam predstavlja vrijeme izvođenja našeg algoritma na ulazu od n elemenata
- Ograničavajući funkciju g(n) možemo shvatiti na sljedeći način:
  - Vrijeme izvođenja našeg algoritma će u najgorem slučaju biti jednako g(n)
  - U svim ostalim slučajevima će biti manje
  - Dajemo garanciju da će naš algoritam uvijek biti jednak ili brži od g(n)
  - **Za najmanju takvu funkciju g(n) ćemo reći da je ona veliko *O***

Odabir ograničenja (1/3)

- Uvijek nam je zadana $T(n)$
- Postoji beskonačno funkcija $g(n)$ koje ograničavaju $T(n)$
- Mi ćemo u ovom kolegiju od te beskonačnosti izdvojiti nekoliko funkcija koje nam dolaze u obzir:
  - $g(n) = 1$
  - $g(n) = \log n$
  - $g(n) = n$
  - $g(n) = n \log n$
  - $g(n) = n^2$, $g(n) = n^3$, $g(n) = n^4$, …
  - $g(n) = 2^n$
  - $g(n) = n!$

Odabir ograničenja (2/3)

- Za dovoljno velike n vrijede sljedeći odnosi:
  - $1 < \log n < n < n \log n < n^2 < n^3 < \ldots < 2^n < n!$



Odabir ograničenja (3/3)

- Obično više funkcija iz navedenog skupa ograničavaju $T(n)$
  - Uvijek ćemo birati **najmanju** ograničavajuću funkciju $g(n)$ i reći da je ta funkcija ograničavajuća za $T(n)$, odnosno da je to veliko O
  - Pisat ćemo: $T(n) = O(g(n))$

Primjer odabira ograničenja

- Imamo zadanu funkciju $T(n) = n + 5$
  - Ograničavaju je sve navedene funkcije osim $g(n) = 1$ i $g(n) = \log n$
  - Od svih tih funkcija biramo $g(n) = n$ jer je najmanja
  - Pišemo $T(n) = O(n)$ i čitamo:
    - "Vrijeme izvođenja algoritma je ograničeno funkcijom n", ili
    - "Porastom broja ulaznih podataka, vrijeme izvođenja algoritma će u najgorem slučaju rasti linearno s brojem ulaznih podataka"
      - Dakle, ako broj ulaznih podataka utrostručimo, utrostručit će se i vrijeme obrade
      - Računamo na to da će u realnim situacijama vrijeme izvođenja ipak biti manje nego u najgorem slučaju

Pravila kod odabira ograničenja

- Ako je $T(n)$ polinom stupnja **r**, tada je ograničavajuća funkcija $\boldsymbol{n^r}$, tj.
  - Poništavamo članove nižeg reda i konstantne članove
  - Izostavljamo konstantu uz član najvišeg reda
    - Primjerice, ako je vrijeme izvođenja prikazano funkcijom $T(n) = 6n^4 - 2n^3 + 5$, koliko je veliko O?
      - Nakon poništavanja članova nižeg reda i konstantnog člana ostaje $6n^4$
      - Izostavljanjem konstante uz član najvišeg reda, dobivamo da je veliko O jednako n^4 i pišemo: $T(n) = 6n^4 - 2n^3 + 5 = O(n^4)$
- $n!$ je jači od svih ostalih članova
- 2^n je jači od svih ostalih članova, osim od $n!$

Primjeri ograničenja

1. Ako je vrijeme izvođenja $T(n) = (2n + 2)^2$, izračunajte O.
   - Odgovor: $T(n) = O(n^2)$
2. Ako je vrijeme izvođenja $T(n) = n^3 + 14n$, izračunajte O.
   - Odgovor: $\mathrm{T(n)} = \mathrm{O}(n^3)$
3. Ako je vrijeme izvođenja $T(n) = 2^n + n^9 + 851$, izračunajte O.
   - Odgovor: $\mathrm{T(n)} = \mathrm{O}(2^n)$
4. Ako je vrijeme izvođenja $T(n) = 2^n + 2n! + 4$, izračunajte O.
   - Odgovor: $\mathrm{T(n)} = \mathrm{O}(n!)$

Realni odnosi između funkcija ograničenja

- Da bismo ilustrirali realne odnose između raznih funkcija ograničenja, poslužit ćemo se tablicom iz priručnika
  - Tablica prikazuje vremena izvođenja za algoritme razne složenosti s istim brojem ulaznih podataka

	$T(n) = n$	$T(n) = n\log(n)$	$T(n) = n^2$	$T(n) = n^3$	$T(n) = 2^n$
$n = 5$	0,005 µs	0,01 µs	0,03 µs	0,13 µs	0,03 ms
$n = 50$	0,05 µs	0,28 µs	2,5 µs	125 µs	13 dana
$n = 100$	0,1 µs	0,66 µs	10 µs	1 ms	4×10^{13} godina

- Tablica pokazuje da je uvijek bolje koristiti algoritme manje složenosti jer su brži

Primjer složenosti algoritma za funkciju 2^n

- Pogledajmo koliko je zapravo spora funkcija 2^n ako uzmemo da je $n = 100$
  - Potrebno je obaviti 2^{100} logičkih operacija
  - Pretpostavimo da imamo računalo koje može obaviti 10^{12} logičkih operacija u 1 sekundi
  - Potrebno vrijeme za izvođenje algoritma je
    - $2^{100} / 10^{12}$ sekundi
    - $= 4 * 10^{10}$ godina
    - $> 4.5 * 10^{9}$ godina
    - Više od procijenjene starosti Zemlje ☺

Izračun vremena izvođenja

- Do sada smo naučili izračunati veliko *O* ako smo imali zadanu funkciju vremena izvođenja *T*(*n*)
  - Odakle nam *T*(*n*)?
- Za izračun *T*(*n*) koristit ćemo *a priori* analizu algoritama
  - To je često više procjena vremena izvođenja nego točna brojka
  - Radimo je nezavisno od računala, operacijskog sustava, programskog jezika i prevoditelja
  - Za izražavanje trajanja izvođenja često umjesto vremena koristimo broj logičkih operacija
    - Kažemo da je brži onaj algoritam koji će se izvršiti u manje logičkih operacija

Primjer izračuna vremena izvođenja (1/2)

- Primjeri izračuna:
  - `int a = 55;`
    - Izvršit će se u 1 logičkoj operaciji i njegova složenost je *O*(1)
  -
```
for (int i = 1; i <= n; i++) {
    x = x * 2;
}
```
    - Unutarnja petlja se sastoji od 1 naredbe, a izvršit će se *n* puta, dakle, T(*n*) = *n*
      - To nam je i intuitivno jasno jer što je veći *n*, točno toliko će nam dulje trajati izvođenje programa
      - Jesmo li što izostavili?
    - Kažemo da je njegova složenost *O*(*n*)

Primjer izračuna vremena izvođenja (2/2)

```
cin >> n;
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= n; j++) {
     cout << i + j;
  }
}
```

- Unutarnja petlja će jednom inicijalizirati varijablu `j` i provjeriti uvjet, a zatim će:
  - n puta provjeriti uvjet u `for`-u i n puta će se povećati brojač u `for`-u
  - n puta će se napraviti zbrajanje i n puta će se napraviti ispis
  - Ukupno trajanje unutarnje petlje je $n + n + n + n + 1 + 1 = 4n + 2$
- Vanjska petlja se sastoji od $2n + 2$ za `for`, te od n puta ponovljene unutarnje petlje, dakle, ukupno: $2n + 2 + n*(4n + 2)$
- Uz prvu liniju imamo $T(n) = 2n + 4 + 4n^2 + 2n = 4n^2 + 4n + 4$
- Kažemo da je njegova složenost $O(n^2)$

Primjeri algoritama različite složenosti (1/2)

- $O(1)$
  - Algoritam koji radi jednako bez obzira što je ulaz
  - Primjerice, algoritam koji provjerava je li broj neparan
- $O(\log n)$
  - Algoritmi utemeljeni na binarnom stablu su često $O(\log n)$
    - Dobro izbalansirano stablo ima $\log n$ razina pa traženje bilo kojeg elementa zahtjeva prolaženje samo kroz jedan čvor na svakoj razini
- $O(n)$
  - Zahtijeva samo jedan prolaz na cijelom ulazu
  - Primjerice, algoritam linearnog traženja koji traži element ispitivanjem svakog elementa polja je $O(n)$ složenosti

Primjeri algoritama različite složenosti (2/2)

- $O(n\log n)$
  - Dobri algoritmi za sortiranje imaju ovu složenost
- $O(n^2)$
  - Primjerice, bubble-sort algoritam za sortiranje
- $O(2^n)$
  - Eksponencijalno vrijeme porasta, vrlo opasno po performanse
  - Koristi se za, primjerice, faktorizaciju velikih brojeva
    - Nema pametnijeg rješenja

NOTACIJA VELIKO Ω

Notacija veliko Ω

- Veliko *O* označava funkciju koja odozgo ograničava $T(n)$
- Veliko Ω označava funkciju koja **odozdo** ograničava $T(n)$
  - Uvijek uzimamo najveću takvu funkciju i pišemo: $T(n) = \Omega(g(n)) \Leftrightarrow$ vrijeme izvođenja je odozdo ograničeno sa $g(n)$
- Veliko Ω je donja granica vremena izvođenja algoritma
  - Niti u kojem slučaju izvođenje ne može biti kraće od Ω
  - Predstavlja najbolji slučaj

Primjeri notacije veliko Ω

- Pogledajmo nekoliko primjera:
  - Algoritam množenja dvije matrice od *n* x *n* elemenata uvijek traje n^2 pa pišemo $T(n) = \Omega(n^2)$
    - U ovom slučaju vrijedi i $T(n) = O(n^2)$, ali općenito ne mora vrijediti
  - Algoritam zbrajanja *n* brojeva uvijek traje *n* pa pišemo $T(n) = \Omega(n)$
    - I u ovom slučaju vrijedi $T(n) = O(n)$
  - Algoritam pronalaska nekog broja među *n* nesortiranih brojeva ima sljedeće vrijednosti:
    - $T(n) = \Omega(1)$
    - $T(n) = O(n)$

A POSTERIORI ANALIZA

Algebra
visoka škola za
primijenjeno računarstvo

Uvod

- Osmislili smo algoritam i implementirali ga u program
- ***A posteriori*** analiza predstavlja analizu izvođenja programa na nekom stvarnom računalu
- Umjesto logičkih operacija koristimo vrijeme
- Mjerimo duljinu izvođenja programa
  - Što je kraće vrijeme izvođenja, program smatramo boljim

Algebra
visoka škola za
primijenjeno računarstvo

Načini mjerenja brzine izvođenja kôda

- Možemo koristiti jednu od sljedećih metoda:
  - Ugrađena struktura **`timeb`** i funkcija **`void _ftime(timeb* time);`**
    - Definirana u standardu i svuda dostupna (**`<sys\timeb.h>`**)
    - Problem je preciznost jer daje vrijeme u milisekundama
  - Windows API funkcija **`QueryPerformanceCounter`**
    - Problem je što radi samo na Windows operacijskim sustavima
    - Više tisuća puta preciznija od prethodne metode jer pristupa fizičkim brojačima u procesoru
    - Potpis funkcije:

```
BOOL QueryPerformanceCounter(LARGE_INTEGER*
lpPerformanceCount);
```