STRUKTURE PODATAKA I ALGORITMI

Predavanje 03

Ishod 1

1

ANALIZA SLOŽENOSTI ALGORITAMA

Strana ▪ 2

2

Uvod

- Algoritam je dobro definiran postupak koja uzima ulaze i pretvara ih u izlaze
  - Algoritam se može pisati na papiru, u našem izmišljenom jeziku (pseudokôdu), nekom programskom jeziku, može se crtati, …

```
BUBBLESORT(A)
1. for i = i to A.length – 1
2.     for j = A.length downto i + 1
3.         if A[j] < A[j – 1]
4.             exchange A[j] with A[j – 1]
```

Primjer iz Introduction to Algorithms, 3rd Edition (Cormen et al)

  - Algoritam implementiramo u nekom programskom jeziku
- Cilj analize složenosti algoritama jest za svaki algoritam reći a) koliko je brz te b) kako se ponaša kad raste broj elemenata koje treba obraditi
  - Na osnovu toga onda donosimo odluku o njegovom korištenju

Strana ▪ 3

ALGEBRA

3

Primjer

- Koliko je brz sljedeći algoritam:

```
for (int i = 0; i < n; i++) {
    if (brojevi[i] == val) {
        cout << "Pronasao!" << endl;
        break;
    }
}
```

- Dva načina analize:

1. *A priori* analiza: procjena/predviđanje brzine izvršavanja
   - Za ovo nam je dovoljan algoritam na papiru
2. *A posteriori* analiza: mjerenje trajanja izvršavanja
   - Za ovo nam treba gotov program na računalu

Strana ▪ 4

ALGEBRA

4

A PRIORI ANALIZA

Vrijeme izvršavanja

- **Vrijeme izvršavanja** (engl. *running time*) je vrijeme potrebno da izvođenje algoritma dođe do kraja
  - Označavat ćemo ga sa ***T*(*n*)**, gdje je *n* broj ulaznih podataka
  - Izražavat ćemo ga u broju operacija, a ne sekundi
    - Manji broj operacija = veća brzina
  - Primjerice, T(1000) = 2981 znači da trajanje algoritma na 1000 ulaznih podataka iznosi 2981 operaciju

Izračun broja operacija (1/2)

- Da bismo mogli procijeniti vrijeme izvršavanja, moramo prebrojati koliko operacija će trebati obaviti
- Uzet ćemo prethodni primjer i krenuti brojati:

1. Inicijalizacija varijable u `for` petlji: 1 operacija
2. Provjera je li `i < n`: 1 operacija
3. Provjera uvjeta uz `if`: 1 operacija
4. Ako je uvjet zadovoljen:
   - Ispis: 1 operacija
   - Izlazak iz petlje: 1 operacija
5. Ako uvjet nije zadovoljen:
   - Povećanje varijable `i`: 1 operacija
   - Odi na korak 2

Strana ▪ 7

7

Izračun broja operacija (2/2)

- Koliko je onda to ukupno operacija? Što nedostaje?
- Nedostaje nam podatak koliko puta će se petlja izvršiti?
  - Nije svejedno hoće li se izvršiti 6 ili 6.000.000 puta...

Strana ▪ 8

8

Tipovi analiza

- Da bismo mogli prebrojati operacije, moramo odlučiti koji tip analize želimo raditi:
  - Najbolji slučaj (engl. *best case scenario*)
  - Najgori slučaj (engl. *worst case scenario*)
  - Srednji/prosječni slučaj (engl. *average case scenario*)

Strana ▪ 9

9

Analiza najboljeg slučaja

- Naš algoritam traži broj `val` u polju
- Kakvi podaci u polju moraju biti da bismo pronašli `val` najbrže moguće?
  - `val` mora se nalaziti odmah na indeksu 0
- Kakvi god bili ulazni podaci i koliko god ih bilo, naš algoritam nikada neće raditi brže od najboljeg slučaja
  - Kažemo da je to gornja granica brzine rada našeg algoritma, tj. naš algoritam ne može biti brži od toga
- U najboljem slučaju će brzina izvršavanja biti: $T(n) = 5$
  - Primijetimo da uopće ne ovisi o veličini polja n
- Analiza najboljeg slučaja se rjeđe koristi jer se u praksi rijetko dešava najbolji slučaj

Strana ▪ 10

10

Analiza najgoreg slučaja

- Kakvi podaci u polju moraju biti da bismo pronašli broj `val` najkasnije moguće?
  - Broj `val` uopće ne smije postojati u polju
- Kakvi god bili ulazni podaci i koliko god ih bilo, naš algoritam nikada neće raditi sporije od najgoreg slučaja
  - Kažemo da je to donja granica brzine rada našeg algoritma
  - Naš algoritam ne može biti sporiji od toga
- U najgorem slučaju će brzina izvršavanja biti: $T(n) = 3n + 1$
  - Primijetimo linearnu ovisnost o veličini polja *n*
- Analiza najgoreg slučaja se vrlo često koristi u praksi
  - Ako ste vi prodavač svog algoritma, ovo je garancija kupcu

Strana ▪ 11

ALGEBRA

11

Analiza srednjeg slučaja

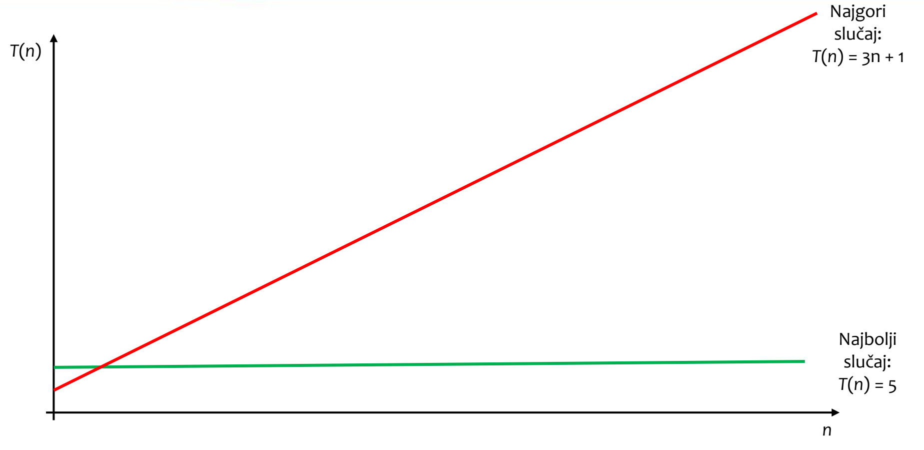
- Razumno je pretpostaviti da ćemo broj `val` pronaći u prosjeku na polovici polja
  - Nekad ćemo ga naći bliže početku, nekad bliže kraju ili ga uopće nećemo naći
- U srednjem slučaju će brzina izvršavanja biti: $T(n) = 3n/2 + 3$
  - Primijetimo ponovno linearnu ovisnost o veličini polja *n*
- Međutim, ako naša početna pretpostavka nije točna, niti srednji slučaj nije točno izračunat
  - U praksi je obično teško izračunati srednji slučaj
  - Često imaju jednaku složenost kao i najgori slučaj
  - U nastavku nećemo promatrati srednje slučajeve

Strana ▪ 12

ALGEBRA

12

Grafički prikaz najboljeg i najgoreg slučaja



- Naš algoritam će garantirano biti ispod gornje i iznad donje crte, bez obzira koliko velik bio *n* i kakvi ulazni podaci bili
  - Nakon određene točke n_0

Pojednostavljivanje prebrojavanja operacija

- Za analizu složenosti nam nije važan točan broj operacija, već je važno shvatiti njihovu ovisnost o *n*
- Analizirajmo najbolji i najgori slučaj algoritma <u>približnim</u> brojenjem operacija i nacrtajmo graf za *n* jednak 1, 10 i 100:

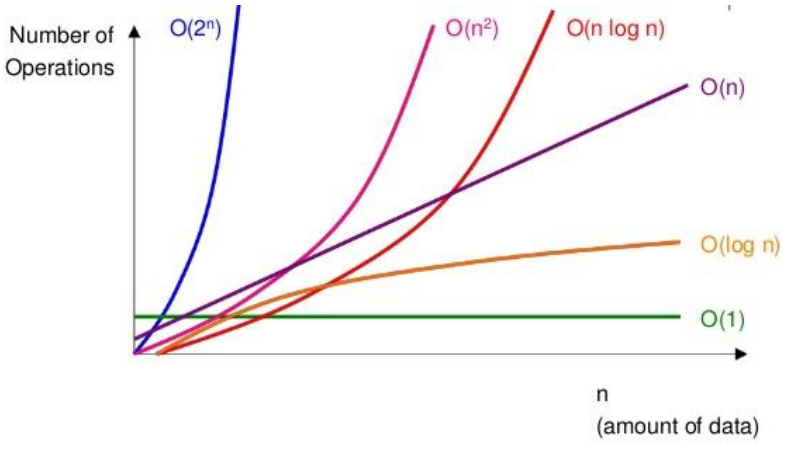
```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        cout << i << " * " << j << " = " << i * j << endl;
    }
}
```

- Primijetimo da vrijeme izvršavanja kvadratno raste s povećanjem broja ulaznih podataka
  - Pri tome nam je potpuno svejedno hoće li biti $T(1000) = 1.000.000$ ili $T(1000) = 1.000.094$

Predefinirane funkcije

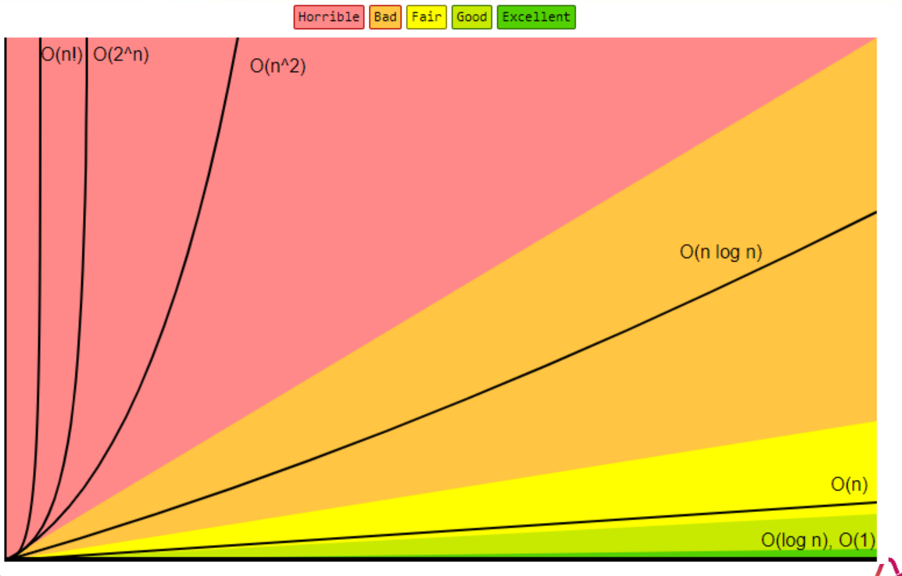
- Ako jedan algoritam ima u najgorem slučaju $\mathrm{T}(n) = \left(\frac{n+4}{2n-7}\right)^{n+1}$, a drugi $\mathrm{T}(n) = \frac{3n+\sqrt{n^2-4n-7}}{2n+3}$, koji je brži?
- Kako bismo lakše uspoređivali algoritme, koristit ćemo nekoliko jednostavnih predefiniranih funkcija
  - $f(n) = 1$
  - $f(n) = \log n$
  - $f(n) = n$
  - $f(n) = n \log n$
  - $f(n) = n^2$ (ili n^3, n^4, …)
  - $f(n) = 2^n$
  - $f(n) = n!$

Odnos funkcija – pogled 1



(C) 2010 Thomas J Cortina, Carnegie Mellon University

Odnos funkcija – pogled 2



Strana ▪ 17

Preuzeto s bigocheatsheet.com

ALGEBRA

17

Označavanje najboljeg i najgoreg slučaja

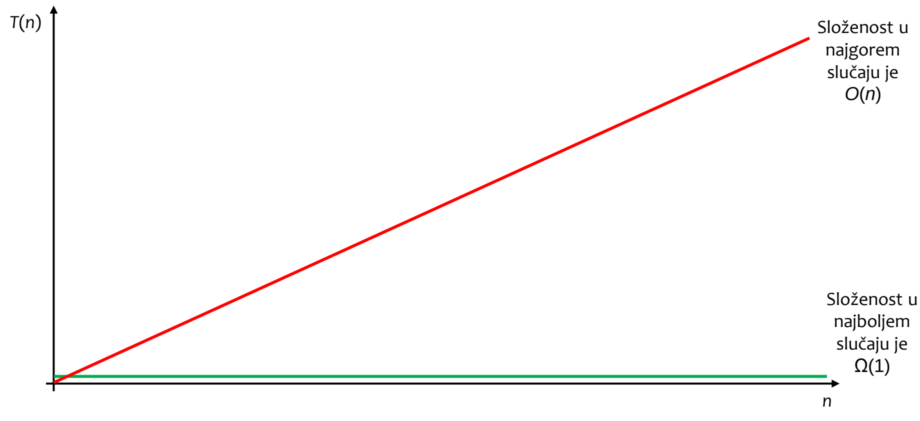
- Kako bismo opisali naš algoritam na razumljiv, standardni način, postupamo ovako:
  - Najbolji slučaj: uzimamo najbližu funkciju f_1 koja je još ispod našeg vremena izvršavanja i kažemo da naš algoritam ima složenost $\Omega(f_1)$
  - Najgori slučaj: uzimamo najbližu funkciju f_2 koja je još iznad našeg vremena izvršavanja i kažemo da naš algoritam ima složenost $O(f_2)$
- Kažemo da koristimo skupinu notacija poznatu pod nazivom Bachmann-Landau notacije ili **asimptotske notacije**

Strana ▪ 18

ALGEBRA

18

Standardni prikaz najboljeg i najgoreg slučaja



- Naš algoritam će garantirano biti ispod gornje i iznad donje crte, bez obzira koliko velik bio n i kakvi ulazni podaci bili
  - Nakon određene točke n_0

Strana ▪ 19

19

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 $n!$

Strana ▪ 20

20

Složenost operacija standardnih kontejnera

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)

Strana ▪ 21

Preuzeto s bigocheatsheet.com

ALGEBRA

21

Složenost standardnih algoritama sortiranja

Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))

Strana ▪ 22

Preuzeto s bigocheatsheet.com

ALGEBRA

22

Zadaci

- Zadaci iz *a priori* analize složenosti algoritama će se temeljiti na tumačenju prethodne dvije tablice
  - Praktično gledano, od svakog programera se to i očekuje
  - Često pitanje na razgovoru za posao
- Primjeri zadataka (argumentirajte):
  1. Ako želite što brže prosječno umetanje, koji kontejner ćete odabrati?
  2. Je li u najgorem slučaju brži BUBBLESORT ili HEAPSORT?
  3. Koji kontejner prosječno najbrže pronađe traženu vrijednost?

A POSTERIORI ANALIZA

Uvod

- Osmislili smo algoritam i implementirali ga u program
  - Procijenili smo i kakva je njegova složenost
- ***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

Strana ▪ 25

25

Imenski prostori

- Tipovi podataka se mogu logički organizirati u imenske prostore (engl. *namespace*)
  - Tip podataka `string` je u imenskom prostoru `std`
  - Tip podataka `Pravokutnik` nije u imenskom prostoru
- Imenski prostori mogu biti ugniježđeni jedan u drugome
- Prilikom korištenja tipa podataka imamo dvije opcije:
  1. Pomoću „`using namespace`” unesemo cijeli imenski prostor pa onda tip podataka koristimo po imenu
  2. Izostavimo „`using namespace`” pa ispred imena tipa podataka stavljamo prefiks imenskog prostora

Strana ▪ 26

26

Primjer

- Primjer opcije 1:

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

int main() {
    string ime = "Mirko";
    cout << ime << endl;
    return 0;
}
```

- Isti program, samo s opcijom 2:

```
#include <iostream>
#include <string>

int main() {
    std::string ime = "Mirko";
    std::cout << ime << std::endl;
    return 0;
}
```


* Primjer vlastitog imenskog prostora

```
namespace moj1 {
   namespace moj2 {
        class Pravokutnik {
        private:
             int sirina;
             int visina;
        public:
             Pravokutnik(int sirina, int visina);
             ~Pravokutnik();
             void inicijaliziraj(int s, int v);
             void mnozi_skalarom(int skalar);
             int povrsina();
             int opseg();
             double dijagonala();
             void iscrtaj();
        };
   }
}
```

Mjerenje brzine izvođenja kôda

- Zaglavlje `<chrono>` sadrži elemente potrebne za mjerenje proteka vremena
  - Svi elementi se nalaze unutar `std::chrono` imenskog prostora, a najvažniji su:
    - Generička klasa `time_point<T>` predstavlja točku u vremenu
      - Parametar T je vrsta sata s kojim radi
    - Klasa `high_resolution_clock` predstavlja sat:
      - Statička* metoda `now()` vraća trenutnu točku u vremenu
    - Generička funkcija `duration_cast<T>` prima razliku dvije točke u vremenu i vraća udaljenost u mjeri T (`seconds`, `milliseconds`, ...)
      - Metoda `count()` vraća `long long` s iznosom trajanja u traženoj mjeri

*Statičke metode zovemo na klasi, a ne na objektu

Primjer korištenja

```
// Spremimo trenutnu točku u vremenu
time_point<high_resolution_clock> t1 = high_resolution_clock::now();

// Odradimo posao
long long s = 0;
for (int i = 0; i < 2100000000; i++) {
    s += i;
}

// Spremimo trenutnu točku u vremenu
time_point<high_resolution_clock> t2 = high_resolution_clock::now();

// Izračunamo razliku dvije točke u vremenu u milisekundama
milliseconds ms = duration_cast<milliseconds>(t2 - t1);
long long trajanje = ms.count();

cout << "Rezultat je: " << s << " u " << trajanje << " ms" << endl;
```

Pakiranje u klasu (1/3)

- Stoperica.h

```
#pragma once
#include <chrono>

class Stoperica {
private:
    std::chrono::time_point<std::chrono::high_resolution_clock> t1;
    std::chrono::time_point<std::chrono::high_resolution_clock> t2;

public:
    void start();
    void stop();
    long long get_elapsed_milliseconds();
};
```


Pakiranje u klasu (2/3)

- Stoperica.cpp

```
#include "Stoperica.h"

void Stoperica::start() {
    t1 = std::chrono::high_resolution_clock::now();
}

void Stoperica::stop() {
    t2 = std::chrono::high_resolution_clock::now();
}

long long Stoperica::get_elapsed_milliseconds() {
    return
        std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1)
                                                              .count();
}
```

Pakiranje u klasu (3/3)

- Source.cpp

```
#include <iostream>
#include "Stoperica.h"
using namespace std;

int main() {
	Stoperica sw;
	sw.start();

	long long s = 0;
	for (int i = 0; i < 2100000000; i++) {
		s += i;
	}

	sw.stop();
	long long trajanje = sw.get_elapsed_milliseconds();

	cout << "Trajanje " << trajanje << " ms" << endl;
	return 0;
}
```


Dodatni materijali

- Dodatni materijali su dostupni na:
  - A priori analysis
    - https://youtu.be/aebaGLFkv54
  - A posteriori analysis
    - https://youtu.be/pH_BbMnaSUo