PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET METROPOLITAN BEOGRAD

KI201 - JAVA 4: STRUKTURE PODATAKA I ALGORITMI – DEO A

Razvoj efikasnih algoritama

Lekcija 05

PRIRUČNIK ZA STUDENTE

KI201 - JAVA 4: STRUKTURE PODATAKA I ALGORITMI – DEO A

Lekcija 05

RAZVOJ EFIKASNIH ALGORITAMA

- Razvoj efikasnih algoritama
- Poglavlje 1: Merenje efikasnosti korišćenjem Veliko O notacije
- Poglavlje 2: Analiza vremenske složenosti algoritama
- Poglavlje 3: Dinamičko programiranje i Fibonačijevi brojevi
- Poglavlje 4: Određivanje NZD primenom Euklidovog algoritma
- Poglavlje 5: Efikasni algoritmi za određivanje prostih brojeva
- Poglavlje 6: Najbliži par tačaka – Primena algoritma „podeli i osvoji“
- Poglavlje 7: Bektreking - Resavanje problema 8 kraljica
- Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

Uvod

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upoznaju sa osnovnim pojmovima u vezi analize algoritama i razvoja efikasnog algoritma:

- Složenost algoritma i načini merenja složenosti algoritama.
- Asimptotska složenost i Veliko O notacija.
- Analiza osnovnih numeričhih algoritama.
- Analiza i implementacija efikasnih numeričkih i geometrijskih 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

Merenje efikasnosti korišćenjem Veliko O notacije

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

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).

Prednosti algoritama uključuju mogućnost da efikasan algoritam može da reši problem uz ograničenja postojećih resursa.

Analizirajmo jedan prost i uvodni primer pre nego što uvedemo standardnu terminologiju. 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.

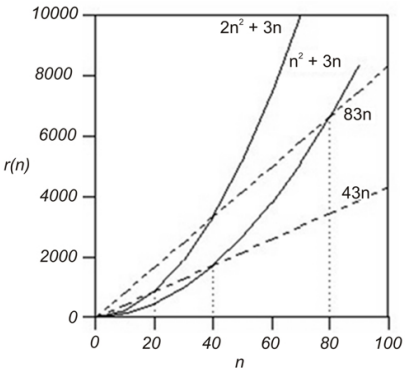
Program A

- 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-2) vreme izvršavanja programa A raste mnogo brže sa povećanjem broja ulaznih podataka *n*.



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

VREMENSKA SLOŽENOST

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

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.

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

Sistem u velikom broju slučajeva zahteva od korisnika da obezbedi gornju granicu na vreme izvršavanja programa. Osim toga, poznato je da postoji veliki broj softvera koji moraju biti dovoljno brzi i obično zahtevaju da se odgovori na određena pitanja daju u realnom vremenu. 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 koraka

Operacija 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

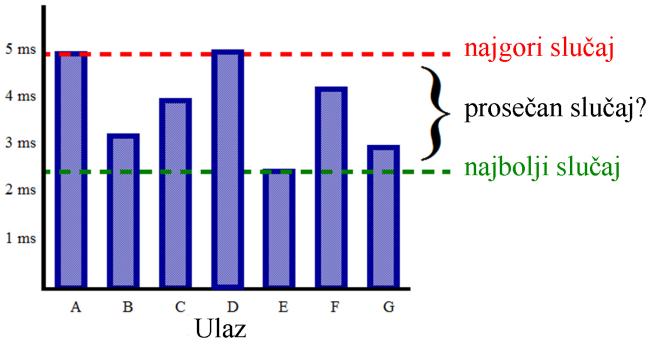
Stoga: ukupan broj poređenja je *n*.

U algoritmima sortiranja, pretraživanja i njima sličnim veličina ulaza će nama biti broj elemenata niza koji treba sortirati ili pretražiti. U nekim drugim algoritmima, na primer Euklidovom algoritmu za određivanje najvećeg zajedničkog delioca, vremenska složenost će zavisiti od veličine brojeva.

NAJBOLJI, NAJGORI I PROSEČNI 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 2 Najbolji, prosečni i najgori slučajevi složnosti

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

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:

```
//  NAREDBE PROGRAMA  BROJ PONAVLJANJA
//-----------------------------------------
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 $\boldsymbol{O(n^2)}$)
- opisuje ponašanje funkcije sa limitom (kada vrednost teži beskonačnosti)

TIPOVI NOTACIJA ASIMPTOTSKE SLOŽENOSTI

Postoje tri tipa notacija kada je u pitanju analiza vremenske složenosti i to: Veliko O, Veliko Omega i Veliko Teta notacija

Postoje sledeći tipovi notacija kod asimptotske složenosti:

a) Asimptotska gornja granica - Veliko O notacija

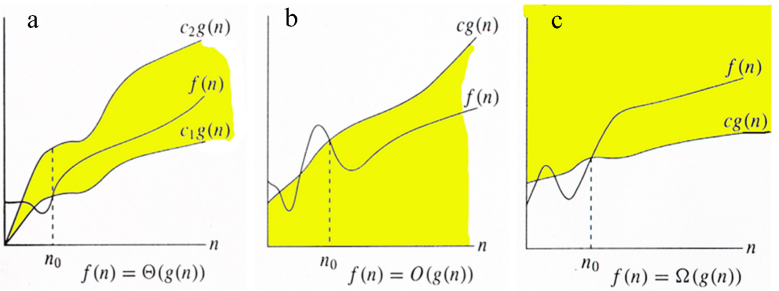
- f(n) je O(g(n)) ako postoji pozitivna konstanta c i $n_0$tako da je f(n)≤ c * g(n) za sve n ≥ n_0
- Formalno, O(g(n)) = f(n): ako ∃ pozitivne konstante c i n_0 tako da je f(n) ≤ c * g(n) ∀ n ≥ n_0

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.

c) Asimptotski opseg mogućih slučajeva – Veliko Teta 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
- f(n) = Θ(g(n)) akko f(n) = O(g(n)) i f(n) = Ω(g(n)).



Slika 1.3 1 Poređenje notacija: a) Teta notacija, b) Veliko O notacija, c) Omega notacija

DEFINICIJA I UPOTREBA VELIKO O NOTACIJE

Veliko O notacija predstavlja gornju granicu odnosno najgore vreme izvršavanja programa

Veliko O je skup svih funkcija čiji je stepen rasta isti ili manji od g(n).

Primer 1: 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 2. 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 ≤ = $3n^3$+ $4n^3$= $7n^3$

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))

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

PRIMERI ODREĐIVANJA 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- 3) .

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

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

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

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

Slika 1.4 Matematička sumiranja koja se koriste kod Veliko O notacije

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 kompleksnosti
  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 $O(n^2)+O(1)=O(n^2)$.

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 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 $n = 50$.

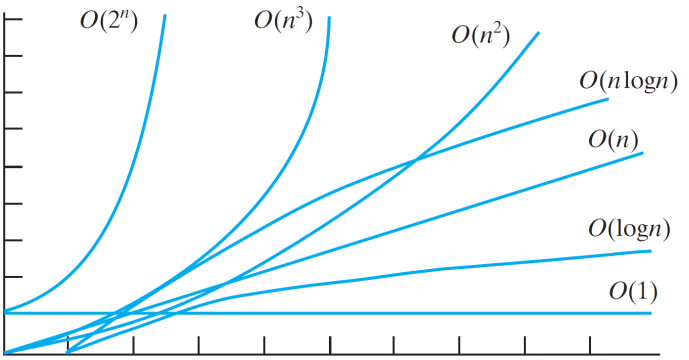
Slika- 2 prikazuje neke korišćene funkcije rasta i promene funkcija rasta za n od $n = 25$ do $n = 50$.

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 1.5 Promena stope rasta raznih vrsta algoritama

Funkcije su prema vremenu izvršavanja uređene na sledeći način, kako je i ilustrovano (Slika-3).

O(1)>O(logn)> O(n)> O(nlogn)>O(n^2)>O(n^3)>O(2^n)



Slika 1.6 Funkcija raste kako se n povećava.

ANALIZA ALGORITAMA - PRIMERI 1-2

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 promeljive, za ispis vrednosti promenjive. Takve korake, sa konstantnim vremenom, ćemo ocenjivati sa O(1).

Primer 1: Oceniti vremensku složenost sledećeg algoritma:

```
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:

```
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

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:

```
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:

$T(n) = c + 2c + 3c + 4c + \ldots + nc$ = c $n(n + 1)/2 = O(n^2)$

Primer 4: Oceniti vremensku složenost sledećeg algoritma:

```
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

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

Oceniti vremensku složenost sledećeg algoritma:

```
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 kontantna. 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)

ZADACI ZA SAMOSTALNI RAD

Na osnovu materijala uraditi samostalno sledeće primere - 1

Primer 1 : Izbrojati broj iteracija u sledećim petljama:

```
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 1.7 Primer 1

Primer 2 : Koliko zvezdica će biti prikazano ako je n jednako 10?

```
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 1.8 Primer 2

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:

```java
public static void mA(int n) {
  for (int i = 0; i < n; i++) {
    System.out.print(Math.random());
  }
}
```

(a)

```java
public static void mB(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++)
      System.out.print(Math.random());
  }
}
```

(b)

```java
public static void mC(int[] m) {
  for (int i = 0; i < m.length; i++) {
    System.out.print(m[i]);
  }

  for (int i = m.length - 1; i >= 0; )
  {
    System.out.print(m[i]);
    i--;
  }
}
```

(c)

```java
public static void mD(int[] m) {
  for (int i = 0; i < m.length; i++) {
    for (int j = 0; j < i; j++)
      System.out.print(m[i] * m[j]);
    }
}
```

(d)

Slika 1.9 Primer 3

Primer 4: Analizirati sledeći algoritam sortiranja:

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

ZADACI ZA SAMOSTALNI RAD - 2

Na osnovu navedenih materijala uraditi samostalno sledeće primere - 2

Primer 5: Za svaki od sledećih isečaka programa:

- 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: Definisati algoritam sa O(n) vremenom izvršavanja za računanje sume brojeva od n1 do n2, gde je n1 < n2. Da li je moguće napraviti O(1) algoritam za isti problem?

Poglavlje 2

Analiza vremenske složenosti algoritama

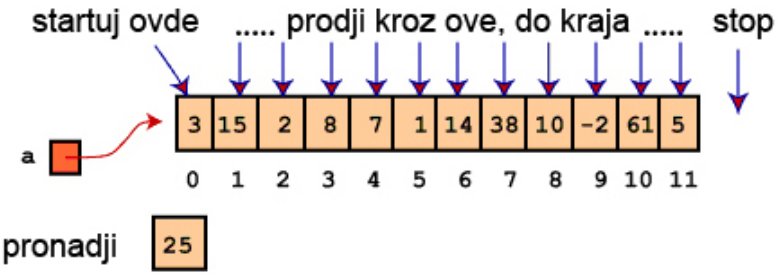
OSNOVI PRETRAŽIVANJA NIZOVA, LINEARNO PRETRAŽIVANJE

Pretraživanje je proces traženja odgovarajućeg elementa u nizu. Osnovni tipovi pretraživanja jesu linearno i binarno pretraživanje. Linearno pretraživanje je algoritam iscrpne pretrage

Pretraživanje je proces traženja odgovarajućeg elementa u nizu, na primer, pronalaženje da li se neki rezultat nalazi u tabeli rezultata. Mnogi algoritmi i strukture podataka su posvećeni pretraživanju, i to će biti detaljnije opisano u lekcijama kao što su heširanje ili stabla pretrage.

Linearno pretraživanje

Pristup linearnog pretraživanja upoređuje ključni element *key* sekvencionalno (redom) sa svim elementima niza. Pretraga se izvršava sve dok se ključ pretrage ne poklopi sa nekim elementom niza, ili dok se ne potroše svi elementi niza, a ne dođe do poklapanja. Ukoliko se desi poklapanje sa ključem, linearno pretraživanje vraća indeks elementa niza koji odgovara ključu pretrage (Slika-1). Ukoliko se ne pronađe ključ u nizu, algoritam vraća vrednost *-1*. Metod linearnog pretraživanja je prikazan u sledećem listingu.



Slika 2.1 Primer linearnog pretraživanja niza

```
public class LinearSearch {
  /** The method for finding a key in the list */
  public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++) {
    if (key == list[i])
        return i;
    }
    return -1;
```

```
  }
}
```

Da bi bolje razumeli postupak, ispratimo sledeće linije koda:

```
int[] list = {1, 4, 4, 2, 5, -3, 6, 2};
int i = linearSearch(list, 4); // Returns 1
int j = linearSearch(list, -4); // Returns -1
int k = linearSearch(list, -3); // Returns 5
```

Metod linearnog pretraživanja upoređuje ključ sa svim elementima niza. Elementi niza mogu biti u bilo kakvom poretku (uređenom tj. sortiranom ili neuređenom). U proseku, algoritam mora da ispita najmanje polovinu elemenata dok ne pronađe ključ, ukoliko on postoji.

Zaključak: Pošto vreme izvršavanja linearnog pretraživanja raste linearno sa brojem elemenata niza, možemo zaključiti da je linearno pretraživanje neefikasan postupak za velike nizove.

UVOD U BINARNO PRETRAŽIVANJE

Binarno pretraživanje eliminiše bar jednu polovinu niza tokom svakog koraka pretrage. Ono predstavlja specijalan tip „podeli i osvoji“ algoritma

Da bi binarno pretraživanje moglo da se primeni, niz vrednosti mora biti sortiran (uređen). Pretpostavimo da je niz koji pretražujemo u rastućem poretku. Binarno pretraživanje prvo upoređuje ključ sa elementom koji je u sredini niza. Uzmimo u obzir sledeća tri slučaja:

- Ukoliko je ključ manji od središnjeg elementa, treba nastaviti pretragu samo u prvoj polovini niza.
- Ukoliko je ključ jednak središnjem elementu, pretraga se završava sa pogotkom.
- Ukoliko je ključ veći od središnjeg elementa, treba nastaviti pretragu samo u drugoj polovini niza.

Binarno pretraživanje eliminiše bar jednu polovinu niza tokom svakog koraka pretrage. Pretpostavimo da niz ima n elemenata. Radi veće ugodnosti deljenja, neka je n oblika n^k. Nakon prvog poređenja ostalo je n/2 elemenata za dalju pretragu; nakon drugog poređenja ostalo je (n/2)/2 elemenata. Nakon k-tog poređenja ostalo je, $n/2^k$ elemenata za dalju pretragu. Kada je $k = \log_2 n$ onda je samo jedan element ostao u nizu, i ostalo je da se obavi samo još jedno poređenje. Stoga, u najgorem slučaju binarnog pretraživanja, potrebno je izvršiti $\log_2 n + 1$ poređenja da bi pronašli element u sortiranom nizu.

Ako poredimo linearno i binarno pretraživanje imamo sledeće zaključke:

- Linearno (sekvencijalno) pretraživanje je korisno da se pretraži ključ u malom nizu ili nesortiranom nizu, ali je neefikasan za velike nizove.
- Binarno pretraživanje je znatno efikasnije, ali zahteva da niz bude unapred sortiran.

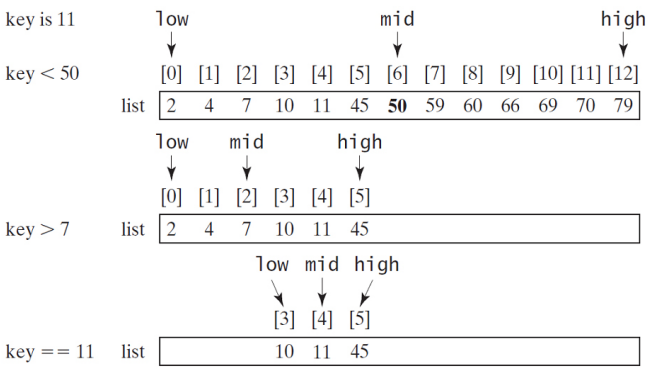
Poređenje binarnog i linearnog (sekvencijalnog) pretraživanja je dato sledećim video snimkom, ali se može naći i na sledećem linku: https://blog.penjee.com/binary-vs-linear-search-animated-gifs/.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ALGORITAM BINARNOG PRETRAŽIVANJA

Linearno pretraživanje je korisno da se pretraži ključ u malom nizu ili nesortiranom nizu, ali je neefikasan za velike nizove. I tim slučajevima prednost ima binarno pretraživanje

Na Slici-2 je prikazan primer pretraživanja ključa **11** u listi elemenata {**2**, **4**, **7**, **10**, **11**, **45**, **50**, **59**, **60**, **66**, **69**, **70**, **79**} korišćenjem binarnog pretraživanja.



Slika 2.2 Binarno pretraživanje eliminiše polovinu niza u svakom sledećem koraku pretrage

Pošto sada znate kako radi binarno pretraživanje, ostalo je da ga implementirate u nekom programskom jeziku. Implementacija se odvija inkrementalno korak po korak. Možete započeti prvu iteraciju pretrage kao što je prikazano na Slici-3a. U ovoj iteraciji se upoređuje ključ sa središnjim elementom u listi čiji je indeks *low* jednak *0* a indeks *high* jednak *list.length() - 1*. Ukoliko je *key < list[mid],* onda treba postaviti indeks *high* na vrednost mid ***- 1*** ;

Ukoliko je *key == list[mid],* izvršen je pogodak i vraćamo indeks *mid*; Ukoliko je *key > list[mid],* postavljamo indeks *low* na vrednost *mid + 1*. Sada razmotrite imlpementaciju metoda dodavanjem petlje, u okviru koje će da se obavi kompletna pretraga, kao što je prikazano na Slici 3b. Pretraga se prekida kada je ključ pronađen, ili kada obavimo sve korake petlje pretrage, tj. kada je *low > high*.

```
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;


    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;



}
```
a

```
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;

  while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;
  }

  return -1; // Not found
}
```
b

Slika 2.3 3 Inkrementalni algoritam binarne pretrage

U slučaju da ključ nije pronađen, indeks *low* postaje tačka insertovanja gde ključ treba biti ubačen kako bi se očuvala uređenost liste. Nekada je mnogo korisnije vratiti tačku ubacivanja nego vrednost *-1*, koja označava da nismo pronašli ključ u listi.

IMPLEMENTACIJA BINARNOG PRETRAŽIVANJA

Implementacija binarnog pretraživanja se odvija inkrementalno korak po korak. Da bi binarno pretraživanje moglo da se primeni, niz vrednosti mora biti sortiran

Šta da uradimo da bi smo istovremeno imali podatak da ključ nije pronađen i mesto gde bi ključ trebalo da bude ubačen da bi lista ostala uređena? Da li da jednostavno vratimo *–low*? Ne! Ukoliko je ključ manji od *list[0],* onda će *low* imati vrednost *0*. Međutim, *-0 je 0* pa bi ovo značilo da se ključ poklapa sa prvim elementom u listi, što naravno nije tačno. Dobar izbor bi bio da metod vrati vrednost ***–low – 1*** ukoliko ključ nije u listi. Vraćenjem vrednosti ***–low – 1*** označavamo ne samo da ključ nije pronađen u listi, već i mesto gde ključ treba biti ubačen u listu kako bi ona ostala sortirana. Kompletan program je dat u nastavku.

```
public class BinarySearch {
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
        high = mid - 1;
    else if (key == list[mid])
        return mid;
    else
        low = mid + 1;
    }

    return -low - 1; // Now high < low
  }
}
```

Prethodni algoritam binarnog pretraživanja vraća indeks niza ukoliko se ključ pretrage nalazi u listi. U suprotnom, algoritam vraća vrednost *–low – 1*.

Šta bi se desilo ako zamenimo uslov *(high >= low)* u liniji 7 sa*(high > low)?* Pretraga će možda omašiti pogodak (slučaj sa samo jednim elementom).

Da li metod radi kada imamo duplikate u listi? Da, sve dok su elementi sortirani u uređenom (i rastućem) poretku, metod vraća indeks jednog od pogođenih elemenata ukoliko se on nalazi u listi.

Da bi bolje razumeli kako radi metod, možete pratiti sledeće iskaze i vrednosti promenljivih ***low*** i ***high*** kada se metod vrati pozivaocu.

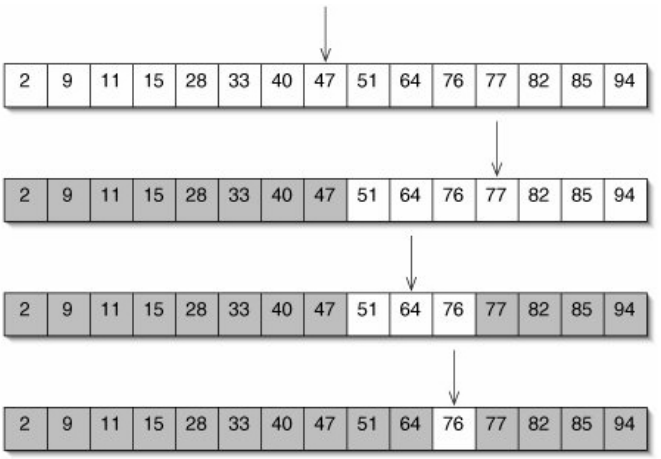
```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
int i = BinarySearch.binarySearch(list, 2); // Low 0, High 1, Returns 0
int j = BinarySearch.binarySearch(list, 11);// Low 3, High 5, Returns 4
int k = BinarySearch.binarySearch(list, 12);// Low 5, High 4, Returns –6
int l = BinarySearch.binarySearch(list, 1); // Low 0, High-1, Returns –1
int m = BinarySearch.binarySearch(list, 3); // Low 1, High 0, Returns –2
```

U prethodnom kodu su izlistane vrednosti indeksa ***low*** i ***high*** kada metod završi sa radom, kao i vrednost koja je rezultat metode.

ANALIZA BINARNOG PRETRAŽIVANJA

Pošto binarno pretraživanje eliminiše polovinu niza nakon svake iteracije pretrage, njegova kompleksnost se može lako odrediti pomoću matematičkog obrasca i iznosi O(logn)

Algoritam binarnog pretraživanja ima za cilj da pretraži da li se ključ pretrage nalazi u sortiranom nizu (Slika-4).



Slika 2.4 Postupak binarnog pretraživanja - pretraga broja 76 u nizu brojeva

Svaka iteracija algoritma sadrži konstantan broj operacija, koji ćemo označiti sa *c*. Neka je sa *T(n)* označena vremenska složenost binarnog pretraživanja liste od *N* elemenata. U cilju generalnosti problema, pretpostavićemo da je *N* stepen broja *2* ($n=2^k$), tj. $k=log_2 n$.

Pošto binarno pretraživanje eliminiše polovinu niza nakon svake iteracije pretrage, složenost se izračunava prema sledećem obascu (Slika-5):

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = T\left(\frac{n}{2^k}\right) + kc \\ &= T(1) + c\log n = 1 + (\log n)c \\ &= O(\log n) \end{aligned}$$

Slika 2.5 Vremenska složenost algoritma binarnog pretraživanja

Ako ignorišemo konstante i nedominantne činioce, kompleksnost binarnog pretraživanja se može napisati kao *O(log n)* .

Jedan algoritam koji ima vremensku složenost O(logn) se naravno naziva logaritamski algoritam.

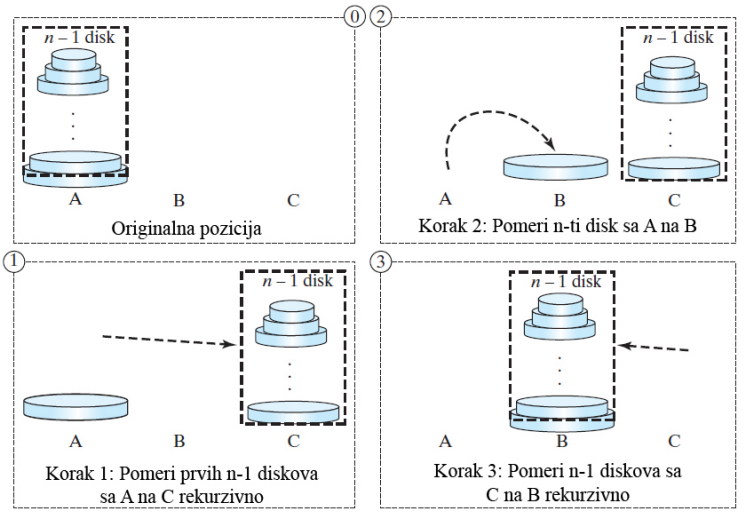
Logaritamski algoritmi rastu sporo sa povećanjem veličine problema. U slučaju binarnog pretraživanja, svaki put kada se duplira veličina niza, najviše jedna operacija poređenja treba biti izvršena. Ukoliko se ulazna veličina niza kvadrira, onda se duplira vreme izvršenja algoritma. Stoga se može zaključiti da su logaritamski algoritmi veoma efikasni.

ANALIZA ALGORITMA HANOJSKIH KULA

Algoritam Hanojskih kula spada u red eksponencijalnih algoritama. Eksponencijalni algoritmi nisu baš praktični za primenu kod problema sa puno ulaznih podataka

U dosadašnjem delu studija, u okviru lekcije o rekurziji, sigurno ste se susreli sa problemom Hanojskih kula, gde se rekurzivno pomera *n* diskova sa tornja *A* na toranj *B* , uz pomoć tornja *C*, na sledeći način (Slika-6):

- Pomeriti prvo *n-1* diskova sa *A* na *C* uz pomoć *B*.
- Pomeriti *n*-ti disk sa *A* na *B*.
- Pomeriti *n-1* diskova sa *C* na *B*, uz pomoć tornja *A*.



Slika 2.6 Postupak algoritma Hanojskih kula

Složenost ovog algoritma se meri brojem ostvarenih pomeranja diskova. Neka sa *T(n)* označimo broj pomeranja u algoritmu koji ima za cilj da pomeri *N* diskova sa tornja *A* na toranj *B*. Očigledno je da je *T(1)* jednako 1. Stoga, u opštem slučaju imamo (Slika-7):

$$
\begin{aligned}
T(n) &= T(n-1) + 1 + T(n-1) \\
&= 2T(n-1) + 1 \\
&= 2(2T(n-2) + 1) + 1 \\
&= 2(2(2T(n-3) + 1) + 1) + 1 \\
&= 2^{n-1}T(1) + 2^{n-2} + \ldots + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 = (2^n - 1) = O(2^n)
\end{aligned}
$$

Slika 2.7 Vremenska složenost algoritma Hanojskih kula

Algoritam složenosti $O(2^n)$ se naravno naziva **eksponencijalni algoritam**.

Eksponencijalni algoritmi nisu baš praktični za probleme sa puno ulaznih podataka. Pretpostavimo da se jedan disk pomera brzinom 1 disk/s. U tom slučaju bi za pomeranje 32 diska bilo neophodno 2^32/(365 * 24 * 60 * 60) = 136 godina, a za 64 diska bi bilo potrebno 2^64/(365 * 24 * 60 * 60) = 585 milijardi godina.

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)

Algoritam iscrpne pretrage, koji pronalazi NDZ dva cela broja m i n, 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*. Algoritam može biti opisan na sledeći način:

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

    return gcd;
}
```

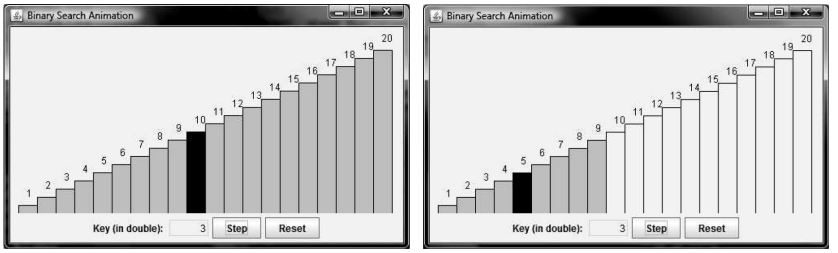
Uz pretpostavku da je m ≥ n, ***for*** petlja će se izvršiti maksimalno *n/2* puta, što u najgorem slučaju skraćuje broj operacija na pola u odnosu na prethodni algoritam. Složenost algoritma je i dalje *O(n)*, ali praktično, ovo je duplo brži algoritam od prethodnog.

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 kompleksnosti, ali je poslednji algoritam očigledno bolji i brži.

ZADATAK 1

Cilj programa je kreiranje animacije binarnog pretraživanja

Napisati aplet koji služi za animaciju algoritma binarnog pretraživanja. Kreirati niz brojeva od 1 do 20 u uređenom poretku. Elementi dijagrama su prikazani u histogramu, kao što je prikazano na Slici- 8. Neophodno je da unesete ključ pretrage u polje za unos (text field). Klikom na dugme *Step* treba da se pokrene deo programa kao u prethodnom zadatku. Koristiti svetlo sivu boju da za dirke onih brojeva koji su u trenutnom opsegu pretraživanja, odnosno crnu boju da za dirku koja predstavlja središnji broj u trenutnom opsegu pretraživanja. Pritisak na dugme Step takođe treba da zamrzne polje kao u prethodnom zadatku. Kada se algoritam završi, prikazati dialog obaveštenje, koji će da obavesti korisnika. Klikom na dugme *Reset* button kreira se novi nasumičan niz za novi početak pretrage. Klikom na ovo dugme polje za unos teksa postaje aktivno (izmenjivo).



Slika 2.8 Progam za prikaz animacije binarnog pretraživanja

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class Exercise22_18 extends Application {
  double radius = 2;
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    HistogramPane pane = new HistogramPane();
    pane.setStyle("-fx-border-color: black");

    Button btStep = new Button("Step");
    Button btReset = new Button("Reset");
    TextField tfKey = new TextField();

    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(
      new Label("Key (in double)"), tfKey, btStep, btReset);
    hBox.setAlignment(Pos.CENTER);

    BorderPane borderPane = new BorderPane();
```

```
    borderPane.setCenter(pane);
    borderPane.setBottom(hBox);

    Label lblStatus = new Label();
    borderPane.setTop(lblStatus);
    BorderPane.setAlignment(lblStatus, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 400, 250);
    primaryStage.setTitle("Exercise22_18"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    StepControl control = new StepControl();
    pane.setNumbers(control.getArray());

    pane.widthProperty().addListener(ov -> pane.repaint());
    pane.heightProperty().addListener(ov -> pane.repaint());

    btStep.setOnAction(e -> {
      if (tfKey.isEditable()) {
        control.setKey(Double.parseDouble(tfKey.getText()));
        tfKey.setEditable(false);
      }

      pane.setColoredBarIndex(control.low, control.high); // Continue
      int status = control.step();

      if (status == 0)
        pane.setColoredBarIndex(control.getCurrentIndex() - 1); // Continue
      else if (status == 1) {
        pane.setColoredBarIndex(control.getCurrentIndex()); // Found
        lblStatus.setText("The key is found in the array at index " +
          control.getCurrentIndex()); // Found
      }
      else if (status == -1)
        lblStatus.setText("The key is not in the array");
    });

    btReset.setOnAction(e -> {
      control.reset();
      tfKey.setEditable(true);
      lblStatus.setText("");
      pane.setColoredBarIndex(0, -1);
    });
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    launch(args);
```

```
}

public final static int ARRAY_SIZE = 20;

class StepControl {
  private int[] list = new int[ARRAY_SIZE];
  private double key = 4.5;

  public int[] getArray() {
    return list;
  }

  public void setKey(double key) {
    this.key = key;
  }

  StepControl() {
    initializeNumbers();
  }

  public void initializeNumbers() {
    for (int i = 0; i < list.length; i++) {
      list[i] = i + 1;
    }
  }

  boolean done = false;

  public int getCurrentIndex() {
    return mid;
  }

  public void reset() {
    low = 0;
    high = list.length - 1;
    done = false;
  }

  int low = 0;
  int high = list.length - 1;
  int mid = (low + high) / 2;

  public int step() {
    if (done) return 1;

    if (low > high)
      return -1;

    mid = (low + high) / 2;
    if (key == list[mid]) {
      done = true;
      return 1;
    }
```

```
      else if (key > list[mid]) {
        low = mid + 1; // Continue to search the second half
        return 0;
      }
      else { // key < list[mid]
        high = mid - 1; // Continue to search the first half
        return 0;
      }
    }
  }

  class HistogramPane extends Pane {
    private int[] numbers;
    private int coloredBarIndex = -1;
    private int low = 0;
    private int high = -1; // Don't display the low and high area initially

    public void setNumbers(int[] numbers) {
      this.numbers = numbers;
      repaint();
    }

    public void setColoredBarIndex(int index) {
      coloredBarIndex = index;
      repaint();
    }

    public void setColoredBarIndex(int low, int high) {
      this.low = low;
      this.high = high;
      repaint();
    }

    public void repaint() {
      // Find maximum integer
      int max = numbers[0];
      for (int i = 1; i < numbers.length; i++) {
        if (max < numbers[i]) {
          max = numbers[i];
        }
      }

      this.getChildren().clear();

      double height = getHeight() * 0.88;
      double width = getWidth() - 20;
      double unitWidth = width * 1.0 / numbers.length;

      int mid = (low + high) / 2;

      for (int i = 0; i < numbers.length; i++) {
        Rectangle bar =  new Rectangle(i * unitWidth + 10, getHeight()
          - (numbers[i] * 1.0 / max) * height, unitWidth, (numbers[i] * 1.0 / max)
```

```
* height);

        if (i <= high && i >= low) {
          bar.setFill(Color.GRAY);
        }
        else {
          bar.setFill(Color.WHITE);
        }

        if (high != -1 && i == mid) {
          bar.setFill(Color.RED);
        }

        bar.setStroke(Color.BLACK);

        this.getChildren().add(bar);
        this.getChildren().add(new Text(i * unitWidth + 10 + 10,
          getHeight() - (numbers[i] * 1.0 / max) * height - 10,
          numbers[i] + ""));
      }
    }
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Proceniti složenost algoritma za sabiranje, kao i algoritma za množenje dve n x m matrice.

Zadatak 2. Definisati algoritam za pronalaženje maksimalnog elementa u nizu. Analizirati složenost algoritma.

Zadatak 3. Definisati algoritam za brisanje duplikata iz niza. Analizirati složenost algoritma.

Zadatak 4. Analizirati složenost sledećeg algoritma za sortiranje.

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

Zadatak 5. Napisati program koji zahteva od korisnika da unese neki string, a zatim ispisuje najduži podstring uzastopnih karaktera koji su poređani rastuće u odnosu na alfabet. Analizirati vremensku složenost programa. Primer:
Ulaz: abacdefkabfh
Izlaz: acdefk

Zadatak 6. Napisati program koji zahteva od korisnika da unese neki string, a zatim ispisuje najdužu podsekvencu karaktera koji su poređani rastuće u odnosu na alfabet. Analizirati vremensku složenost programa.
Ulaz: Welcome **Izlaz**: elm

Zadatak 7. (Podniz istih brojeva) 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:

Ulaz: 2 4 4 8 8 8 8 2 4 4 0

Izlaz:

Najduži podniz istih brojeva počinje na indeksu 3 sa 4 elementa vrednosti 8.

Poglavlje 3

Dinamičko programiranje i Fibonačijevi brojevi

OSNOVI DINAMIČKOG PROGRAMIRANJA

Kada problem pokazuje optimalnu podstrukturu, možemo ga rešiti brzo koristeći dinamičko programiranje - pristup koji izbegava ponovno izračunavanje rešenja koja su već izračunata

U situacijama kada se optimalno rešenje problema može konstruisati iz optimalnog rešenja potproblema, i preklapanjem potproblema, glavni problem možemo rešiti brzo korišćenjem **dinamičkog programiranja**, pristupa koji izbegava ponovno izračunavanje rešenja koja su već izračunata. Sama reč programiranje u terminu dinamičko programiranje se odnosi na popunjavanje tabele pri rešavanju problema, a ne na upotrebu programskog jezika ili računara.

Pregled dinamičkog programiranja

Pojam *optimalna podstruktura* označava traženje optimalnih rešenja potproblema i njihovo korišćenje u cilju traženja optimalnog rešenja celokupnog problema. Na primer, najkraći put do datog čvora (označimo ga kao krajnji) od proizvoljnog čvora u acikličnom grafu, može se izračunati tako što se najpre izračuna najkraći put od krajnjeg čvora do njegovih susednih, zatim isti princip primenimo na njegove susede, itd. Dakle, problem koji ima optimalnu podstrukturu se može rešiti u sledećim koracima:

- Razbiti problem na manje potprobleme.
- Rešiti date potprobleme koristeći ova tri koraka rekurzivno.
- Iskoristiti pronađena optimalna rešenja potproblema kako bi se našlo optimalno rešenje glavnog problema.

- Potprobleme je potrebno deliti na pod-potprobleme, sve dok se ne dođe do jednostavnih slučaja koje je jednostavno rešiti.

Pomoću algoritama dinamičkog programiranja, potproblemi jednog problema postaju međusobno zavisni, pa je dovoljno da ih samo jednom rešite, da bi rad bio lakši i brži.

Primeri dinamičkog programiranja

Najuobičajeni primeri primene algoritma dinamičkog programiranja su: Dijkstra algoritam, Fibonačijeva sekvenca, Algoritmi sa stringovima (najduža zajednička podsekvenca, najduži zajednički podstring, itd...), Belman-Ford algoritam, Flojd-Varšalov algoritam, Problem ranca, neke metode za rešavanje Problema trgovačkog putnika, itd.

PRIMENA DINAMIČKOG PROGRAMIRANJA – FIBONAČIJEVI BROJEVI

Sama reč programiranje u terminu dinamičko programiranje se odnosi na popunjavanje tabele pri rešavanju problema. Na taj način sprečavamo ponovno rešavanje već rešenih potproblema

U Fibonačijevom nizu, na primer, $F_3 = F_1+F_2$ i $F_4 = F_2+F_3$, računanje svakog broja zahteva nalaženje F_2. Kako su F_2 i F_4 potrebni za nalaženje F_5 naivnim pristupom bi za računanje F_5 bilo potrebno nalaženje F_2 nekoliko puta.

Kako bi se ovo izbeglo, potrebno je sačuvati rešenja onih problema koji su već rešeni, kako bi se mogli kasnije iskoristiti. Ovo se zove *memoizacija*. Ukoliko je očigledno da rešeni potproblemi nisu više potrebni, mogu se odbaciti kako bi se sačuvao memorijski prostor. Ovaj algoritam koristi najčešće jedan od sledeća dva pristupa:

- *Odozgo na dole*: Problem se rastavi na potprobleme, potproblemi se reše i pamte se njihova rešenja, za slucaj njihove kasnije upotrebe. Ovaj pristup predstavlja kombinovanje rekurzije i memoizacije.
- *Odozdo na gore*: Svi potproblemi se redom rešavaju i koriste za nalaženje većih. Ovaj pristup je bolji zbog čuvanja memorijskog prostora na steku, ali je ponekad teško odrediti koji su sve potproblemi potrebni za traženje datog.

Naivna implementacija funkcije za traženje n-tog člana Fibonačijevog niza se bazira direktno na matematičkoj definiciji:

```
function fib(n)
    if n = 0 or n = 1
        return n
    else
        return fib(n − 1) + fib(n − 2)
```

Primetimo da, ako se pozove funkcija *fib(5)*, manje funkcije se pozivaju veći broj puta, što raste eksponencijalno. Ako se koristi pristup ***odozgo na dole*** (memoizacija) tada se potproblemi rešavaju tačno jednom, jer se njihove vrednosti pamte:

```
var m := map(0 → 1, 1 → 1)
   function fib(n)
    if n not in keys(m)
        m[n] := fib(n − 1) + fib(n − 2)
    return m[n]
```

Međutim, koristeći pristup *odozdo na gore*, linearna prostorna složenost se može preobraziti u konstantnu:

```
function fib(n)
    var previousFib := 1, currentFib := 1
    repeat n − 1 times
```

```
        var newFib := previousFib + currentFib
        previousFib := currentFib
        currentFib  := newFib
    return currentFib
```

ANALIZA REKURZIVNOG REŠENJA FIBONAČIJEVIH BROJEVA

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

U lekciji o rekurziji, dato je rekurzivno rešenje za određivanje Fibonačijevih brojeva, kao što sledi u nastavku:

```
public static long fib(long index) {
 if (index == 0) // Base case
  return 0;
 else if (index == 1) // Base case
  return 1;
 else // Reduction and recursive calls
  return fib(index - 1) + fib(index - 2);
}
```

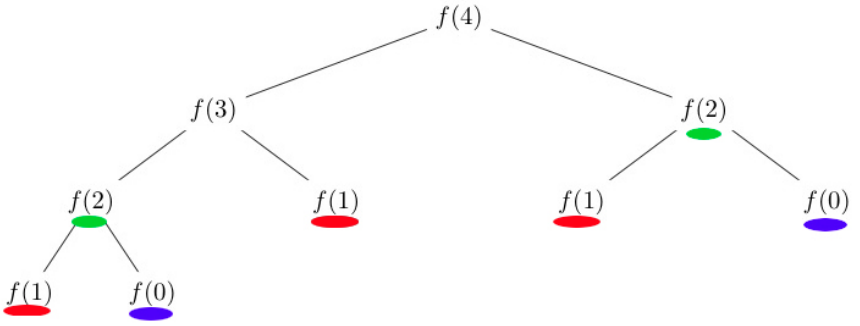
Dokažimo sada da je složenost ovog algoritma $O(2^n)$. Uzećemo da je indeks člana koji određujemo n. Označimo sa $T(n)$ složenost algoritma koji pronalazi $fib(n)$, a sa c označimo konstantno vreme koje je potrebno za poređenje indeksa sa vrednostima 0 i 1. Očigledno, $T(1)$ je c. Stoga, u opštem slučaju imamo (Slika-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 3.1 Vremenska složenost algoritma Fibonačijevih brojeva

Slično kao i u analizi algoritma Hanojskih kula, i ovde možemo pokazati da je složenost algoritma $T(n)= O(2^n)$. Međutim, algoritam nije efikasan.

Problem sa rekurzivnim *fib* metodom je taj što se dešava da se metod pozove 2 ili više puta sa istim parametrima. Da bi smo odredili *fib(4)* , moramo da pozovemo *fib(3)* i *fib(2)*. Da bi odredili *fib(3)* , moramo da pozovemo *fib(2)* i *fib(1)* . Primetimo da je ***fib(2)*** pozvan 2 puta (Slika-2). Stoga, moguće je poboljšati algoritam tako što bi smo izbegli dupliranje poziva metode ***fib*** sa istim argumentom, što će biti prikazano u narednoj sekciji.



Slika 3.2 Rekurzivni postupak određivanja Fibonačijevog broja f(4)

ANALIZA ITERATIVNOG ALGORITMA ODREĐIVANJA FIBONAČIJEVIH BROJEVA

Ovaj algoritam za određivanje Fibonačijevih brojeva koristi princip dinamičkog programiranja. Na taj način izbegavamo dupliranje poziva za iste argumente funkcije

Moguće je poboljšati rekurzivan algoritam da bi izbegli dupliranje poziva metode *fib* sa istim argumentom. Primetimo 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*.

Postupak ažuriranja vrednosti *f0* i *f1*, kao i dobijanje novog člana bi izgledalo kao na sledećoj slici (Slika-3).

```
                      f0 f1 f2
Fibonačijeva serija : 0  1  1  2  3  5  8  13  21  34  55  89 ...
           indeksi : 0  1  2  3  4  5  6  7   8   9   10  11

                         f0 f1 f2
Fibonačijeva serija : 0  1  1  2  3  5  8  13  21  34  55  89 ...
           indeksi : 0  1  2  3  4  5  6  7   8   9   10  11

                                                 f0  f1  f2
Fibonačijeva serija : 0  1  1  2  3  5  8  13  21  34  55  89 ...
           indeksi : 0  1  2  3  4  5  6  7   8   9   10  11
```

Slika 3.3 3 Promenljive f0, f1, i f2 čuvaju tri uzastopna Fibonačijeva broja u seriji.

Ovaj algoritam za određivanje Fibonačijevih brojeva koristi princip *dinamičkog programiranja*. Iterativni metod za određivanje Fibonačijevih brojeva se može napisati na sledeći način:

```
public static long fib(long n) {
    long f0 = 0; // For fib(0)
    long f1 = 1; // For fib(1)
    long f2 = 1; // For 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 $O(2^n)$.

Dinamičko programiranje je proces rešavanja podproblema sa ciljem da se dodje do konačnog rešenja. Ovo prirodno vodi do rekurzivnog rešenja. Međutim, u slučaju Fibonačijevih brojeva, nije pogodno koristiti rekurziju pošto se podproblemi preklapaju. Osnovna ideja dinamičkog programiranja je da se svaki podproblem reši samo jednom, i da se rešenja podproblema sačuvaju kako ne bi smo morali dva puta da rešavamo isti podproblem.

ZADACI ZA SAMOSTALNI RAD

Na osnovu materijala uraditi samostalno sledeći zadatak

Zadatak 1. Napisati program koji će izvršiti merenje vremena izvršavanja rekurzivnog rešenja i rešenja koje koristi dinamičko programiranje za neko veliko N.

Poglavlje 4

Određivanje NZD primenom Euklidovog algoritma

DEFINICIJA I IMPLEMENTACIJA EUKLIDOVOG ALGORITMA

Mnogo efikasniji algoritam za određivanje NZD dva broja je otkrio Euklid oko 300. god. pre nove ere

Jedan od najefikasnijih algoritama za određivanje NZD dva broja je otkrio Euklid oko 300. god. pre nove ere. Ovo je jedan od najstarijih i najpoznatijih algoritama. On može biti rekurzivno definisan na sledeći način. Neka je sa *gcd(m, n)* označen NZD brojeva *m* i *n*:

- Ako je *m % n* jednako *0*, onda je *gcd (m, n)* jednako *n*.
- U suprotnom, *gcd(m, n)* je jednako *gcd(n, m % n)*.

Nije uopšte teško dokazati tačnost ovog algoritma, što je i pokazano na predmetima kao što su *Uvod u objektno programiranje* ili *Diskretne strukture*. Ponovićemo radi obnavljanja gradiva. Pretpostavimo da je *m % n = r*. Stoga, *m = qn + r*, gde je *q* količnik od *m / n*. Bilo koji broj koji je delilac *m* i *n* mora biti i delilac broja r. Dakle, gcd(m, n) će biti isto kao i gcd(n, r), gde je r = m % n.

```
int NZD(int m, int n){
 int pom;
 while(n!=0){
  pom=n;
  n=m%n;
  m=pom;
 }
 return m;
}
```

Implementacija rekurzivnog algoritma je data u nastavku:

```
import java.util.Scanner;
public class GCDEuclid {

  public static int gcd(int m, int n) {
    if (m % n == 0)
    return n;
    else
    return gcd(n, m % n);
```

```
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    System.out.print("Enter first integer: ");
    int m = input.nextInt();
    System.out.print("Enter second integer: ");
    int n = input.nextInt();

    System.out.println("The greatest common divisor for " + m +
    " and " + n + " is " + gcd(m, n));
  }
}
```

Moguće izvršavanje programa:

```
Enter first integer: 2525 (ENTER)
Enter second integer: 125 (ENTER)
The greatest common divisor for 2525 and 125 is 25
```

ANALIZA NAJBOLJEG I NAJGOREG SLUČAJA

U najboljem slučaju kada je m % n jednako 0 algoritam zahteva samo jedan korak da odredi NZD. U najgorem slučaju, vremenska kompleksnost gcd metoda je ustvari O(logn)

U najboljem slučaju, kada je *m % n* jednako *0*, algoritam zahteva samo jedan korak da odredi NZD.

Međutim, malo je teže analizirati prosečan slučaj. Kakogod, možemo dokazati da je vremenska složenost najgoreg slučaja reda O(logn).

Ako pretpostavimo da je *m>=n*, možemo pokazati da je *m % n < m / 2*, na sledeći način:

- Ako je *n <= m / 2*, onda je *m % n < m / 2*, pošto ostatak broja m podeljen sa *n* je uvek manji od *n*.
- Ako je *n > m / 2*, onda je *m % n = m – n < m / 2*. Stoga, *m % n < m / 2*.

Euklidov algoritam rekurzivno poziva metod *gcd*. On prvo poziva *gcd(m, n)*, zatim poziva *gcd(n, m % n)*, pa onda *gcd(m % n, n % (m % n))*, i tako dalje, kao što je prikazano u nastavku:

gcd(m, n)

= gcd(n, m % n)

= gcd(m % n, n % (m % n))

= ...

Pošto je *m % n < m / 2* i *n % (m % n) < n / 2*, argument prosleđen metodi *gcd* se redukuje za polovinu u svakoj sledećoj iteraciji. Nakon dva poziva metode *gcd*, drugi parametar je sigurno manji od n/2. Nakon 4 poziva metode *gcd*, drugi parametar postaje manji od n/4. Nakon šest poziva metode *gcd*, drugi parametar postaje manji od $n/2^3$. Neka sa *k* označimo broj poziva metode *gcd*. Stoga, nakon k-tog poziva metode *gcd*, drugi parametar je manji od $n/2^{k/2}$ što je veće ili jednako 1. Stoga imamo:

$$\frac{n}{2^{k/2}} \geq 1 \Rightarrow n \geq 2^{k/2} \Rightarrow logn \geq k/2 \Rightarrow k \leq 2logn$$

Tako, imamo sledeću nejednačinu k ≤ 2logn, pa možemo da zaključimo da je vremenska složenost *gcd* metoda ustvari O(logn).

Literatura:Liang D., Introduction to Java Programming, 9th edition, 2009.

PRIMER NAJGOREG SLUČAJA

Najgori slučaj nastaje kada za dva broja imamo najveći broj podela i provera. Ispostavlja se da ustvari dva susedna Fibonačijeva broja vode do najvećeg broja podela i provera

Najgori slučaj nastaje kada za dva broja imamo najveći broj podela i provera. Ispostavlja se da ustvari dva susedna Fibonačijeva broja vode do najvećeg broja podela i provera. Podsetimo se da Fibonačijeva serija počinje brojevima 0 i 1, i svaki sledeći član se dobija kao zbir prethodna dva, pa imamo sledeću seriju brojeva: 0 1 1 2 3 5 8 13 21 34 55 89 . . .

Ova serija može rekurzivno biti definisana kao

fib(0) = 0;

fib(1) = 1;

fib(index) = fib(index - 2) + fib(index - 1); index >= 2

Za dva susedna Fibonačijeva broja *fib(index)* i *fib(index - 1)* :

gcd(fib(index), fib(index - 1))

= gcd(fib(index - 1), fib(index - 2))

= gcd(fib(index - 2), fib(index - 3))

= gcd(fib(index - 3), fib(index - 4))

= ...

= gcd(fib(2), fib(1))

= 1

Primer:

```
gcd(21, 13)
= gcd(13, 8)
= gcd(8, 5)
= gcd(5, 3)
= gcd(3, 2)
= gcd(2, 1)
= 1
```

Vidimo da je broj poziva metoda ***gcd*** isti kao i indeks Fibonačijevog broja. Možemo dokazati da važi da je *index* ≤1.44 log*n* pri čemu je *n* = fib(index - 1). Ovo je mnogo jača granica nego *index* ≤log*n*.

Na narednoj Slici- 1 u sumirane složenosti tri algoritma za određivanje NZD dva broja:

Kompleksnost	Opis
$O(n)$	Brute force, provera svih mogućih delilaca
$O(n)$	Provera polovine mogućih delilaca
$O(\log n)$	Euklidov algoritam

Slika 4.1 Poređenje složenosti NZD algoritama

Literatura:Liang D., Introduction to Java Programming, 9th edition, 2009.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Dokazati da je sledeći algoritam za pronalaženje najvećeg zajedničkog delioca (NZD) korektan.

```
int gcd = 1;
for (int k = Math.min(Math.sqrt(n), Math.sqrt(m)); k >= 1; k--) {
    if (m % k == 0 && n % k == 0) {
        gcd = k;
        break;
    }
}
```

Zadatak 2. Napisati program koji računa vreme izvršavanja algoritma za pronalaženje NZD uzastopnih fibonačijevih brojeva od 40. broja do 45. broja koristeći NZD algoritam iz prethodnog zadatka.

Poglavlje 5

Efikasni algoritmi za određivanje prostih brojeva

EFIKASAN ALGORITAM SLOŽENOSTI REDA KORENA

Ako n nije prost, onda n mora da sadrži faktor koji je veći od 1, a manji ili jednak od √n. Na ovaj način dobijamo koreni algoritam za određivanja da li je neki broj prost

Ranije je opisan *algoritam iscrpne pretrage* (brute-force) za određivanje prostih brojeva. Taj algoritam proverava da li brojevi *2, 3, 4, 5, ... ,* i *n/2* dele broj *n* bez ostatka. Ukoliko ne, onda je *n* prost broj. Ovaj algoritam je složenosti *O(n)* .

U stvari, ono što još možemo dokazati je sledeća činjenica:

- Ako *n* nije prost, onda *n* mora da sadrži faktor koji je veći od 1, a manji ili jednak od √n .

Dokaz:

- Pošto n nije prost broj, postoje bar dva broja *p* i *q* takva da je $n = pq$, pri čemu je $1<p \leq q$. Primetimo da je $n = \sqrt{n}\sqrt{n}$. *p* mora biti manji ili jednak od √n. Stoga je dovoljno proveriti da li brojevi 2, 3, 4, 5, . . . , ili √n dele *n* bez ostatka. Ukoliko je dogovor ne, onda *n* nije post. Ovo značajno smanjuje vremensku složenost algoritma na $O(\sqrt{n})$. Sada napišimo program koji pronalazi sve proste brojeve do *n*. Program *main()* je u nastavku. Vremenska složenost programa koji pronalazi sve proste brojeve do *n* je $O(n\sqrt{n})$.

```
import java.util.Scanner;
public class PrimeNumbers {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Find all prime numbers <= n, enter n: ");
    int n = input.nextInt();

    final int NUMBER_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness

    System.out.println("The prime numbers are:");

    while (number <= n) {
      boolean isPrime = true;

      int squareRoot= (int)(Math.sqrt(number));
```

```
      for (int divisor = 2; divisor <= squareRoot;
          divisor++) {
        if (number % divisor == 0) {
          isPrime = false;
          break;
        }
      }

      if (isPrime) {
        count++;

        if (count % NUMBER_PER_LINE == 0)
          System.out.printf("%7d\n", number);
        else
          System.out.printf("%7d", number);
      }

      number++;
    }

    System.out.println("\n" + count +
      " prime(s) less than or equal to " + n);
  }
}
```

POBOLJŠANI ALGORITAM DINAMIČKOG PROGRAMIRANJA

Da bi odredili da li je i prost broj, dovoljno je samo da proverimo da li je i deljivo prostim brojevima između 2 i sqrt(i) . Proste brojeve do sqrt(i) čuvamo u tabeli

Da bi odredili da li je i prost broj, uobičajeni algoritam proverava da li je i deljivo brojevima 2, 3, 4, 5, . . . , i √n. Međutim, dovoljno je samo da proverimo da li je i deljivo prostim brojevima između 2 i √n. Ovo vodi do još efikasnijeg algoritma za određivanje svih prostih brojeva do *n*.

Možemo dokazati da ako *i* nije prost broj, onda mora da postoji prost broj *p* tako da važi i = *pq* i p ≤ q (dokaz je izvan opsega ovog kursa).

Analiza algoritma: Označimo sa *π(i)* broj prostih brojeva koji su manji ili jednaki od i. Ono što možemo da izbrojimo jeste da su prosti brojevi manji od 20: 2, 3, 5, 7, 11, 13, 17, i 19. Stoga, π(2) je 1, π(3) je 2, π(6) je 3, i π(20) je 8. Dokazano je da je π(i) približno jednako i/log i. Za svaki broj i, algoritam proverava da li je i deljivo prostim brojem manjim ili jednakim √n; . Broj prostih brojeva manjih ili jednakih √n je: √i / log √i.

Na osnovu ovih pretpostavki, daljim izvođenjem (u koje ne ulazimo), možemo dobiti da je složenost ovog algoritma O(n√n/logn).

Ovaj algoritam je jedan od primera dinamičkog programiranja. Algoritam čuva rezultate podproblema i to kasnije koristi sa ciljem da odredi da li je novi broj prost.

Kod algoritma je dat u nastavku.

```
import java.util.Scanner;
public class Zadatak6 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Find all prime numbers <= n, enter n: ");
    int n = input.nextInt();

    // A list to hold prime numbers
    java.util.List<Integer> list =
      new java.util.ArrayList<Integer>();

    final int NUMBER_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness
    int squareRoot = 1; // Check whether number <= squareRoot

    System.out.println("The prime numbers are \n");

    while (number <= n) {
      boolean isPrime = true;

      if (squareRoot * squareRoot < number) squareRoot++;

      // ClosestPair if number is prime
      for (int k = 0; k < list.size()
                        &amp;amp;amp;amp;&amp;amp;amp;amp; list.get(k) <=
squareRoot; k++) {
        if (number % list.get(k) == 0) { // If true, not prime
          isPrime = false; // Set isPrime to false
          break; // Exit the for loop
        }
      }

      if (isPrime) {
        count++; // Increase the count
        list.add(number); // Add a new prime to the list
        if (count % NUMBER_PER_LINE == 0) {
          // Print the number and advance to the new line
          System.out.println(number);
        }
        else
          System.out.print(number + " ");
      }
      number++;
    }

    System.out.println("\n" + count +
      " prime(s) less than or equal to " + n);
```

```
  }
}
```

SITO ERATOSTENA

Algoritam radi tako što se uzme niz logičkih vrednosti kao indikator da li je neki broj prost ili ne. Zatim se svi članovi niza deljivi sa 2,3,4,... setuju na false vrednost (nisu prosti)

Eratosten (276–194 B.C.) je bio grčki matematičar koji je izumeo veoma pametan algoritam, poznat kao Sieve of Eratosthenes, za određivanje svih prostih brojeva ≤ n. Njegov algoritam je da se uzme niz pod imenom *primes* koji se sastoji iz *n* logičkih vrednosti tipa *boolean*. Inicijalno, svi elementi niza *primes* su postavljeni na vrednost *true*. Pošto svi brojevi koji se pomnože sa *2* nisu prosti, onda postaviti sve članove *primes[2 * i] na* vrednost *false*, za svako *i*, gde je $2 \le i \le n/2$, kao što je prikazano na sledećoj slici. Pošto su za nas zanemarive vrednost elemenata *primes[0] i primes[1]* , ove vrednosti će biti označene u nastavku sa *X*.

Niz primes

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
initial	×	×	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$k=2$	×	×	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
$k=3$	×	×	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F
$k=5$	×	×	Ⓣ	Ⓣ	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	F	F	F

Slika 5.1 Vrednosti u nizu primes se menjaju nakon obrade svakog sledećeg broja k

Pošto svi brojevi deljivi sa 3 nisu prosti, postaviti članove *primes[3 * i] na vrednost false za svako* $3 \le i \le n/3$. *S obzirom da su takođe svi brojevi pomnoženi sa 5 nisu prosti, postaviti takođe primes[5 * i] na false* za sve $5 \le i \le n/5$. Primetimo da nije neophodno uzeti u obzir brojeve pomnožene sa 4, jer umnošci broja 4 su takođe umnošci broja 2, što je već uzeto u obzir.

Slično tome, nije potrebno vršiti množenje brojevima **6**, **8**, i **9**. Jedino što treba biti uzeto u obzir su umnošci prostih brojeva **k = 2**, **3**, **5**, **7**, **11**, . . . , a odgovarajuće elemente niza *primes* treba setovati na vrednost ***false***. Nakon svih ovih operacija treba proći kroz članove niza *primes*. Ukoliko je ***primes[i]*** i dalje tačno onda je **i** prost broj kao što je i pokazano na prethodnoj slici.

U nastavku je program koji koristi algoritam Eratostenovog sita.

```
import java.util.Scanner;
public class SieveOfEratosthenes {
 public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  System.out.print("Find all prime numbers <= n, enter n: ");
  int n = input.nextInt();

  boolean[] primes = new boolean[n + 1]; // Prime number sieve

  // Initialize primes[i] to true
```

```
  for (int i = 0; i < primes.length; i++) {
   primes[i] = true;
  }

  for (int k = 2; k <= n / k; k++) {
   if (primes[k]) {
    for (int i = k; i <= n / k; i++) {
     primes[k * i] = false; // k * i is not prime
    }
   }
  }

  int count = 0; // Count the number of prime numbers found so far
  // Print prime numbers
  for (int i = 2; i < primes.length; i++) {
   if (primes[i]) {
    count++;
    if (count % 10 == 0)
     System.out.printf("%7d\n", i);
    else
     System.out.printf("%7d", i);
   }
  }

  System.out.println("\n" + count +
    " prime(s) less than or equal to " + n);
 }
}
```

ANALIZA ALGORITMA SITA ERATOSTENA

Kompleksnost ovog algoritma je O(nn/logn) . Međutim stvarna vremenska složenost je znatno bolja od ove vrednosti. Algoritam Eratostenovog sita je veoma dobar za male vrednosti n

Koja je vremenska složenost ovog algoritma?

Za svaki prost broj k, algoritam postavlja *primes[k * i]* na vrednost ***false*** . Ovo se obavlja n / k – k + 1 puta u ***for*** petlji. Stoga, vremenska složenost određivanja svih prostih brojeva do n je:

n/2 - 2 + 1 + n/3 - 3 + 1 + n/5 - 5 + 1 + n/7 - 7 + 1 + n/11 - 11 + 1...

= O(n/2 + n/3 + n/5 + n/7 + n/11 + ...) < O(nπ(n))

= O(n √n / log n)

Gornja granica O(n √n / log n) je veoma labava. Stvarna vremenska složenost je znatno bolja od O(n √n / log n) .

Algoritam Eratostenovog sita je veoma dobar za male vrednosti n, tako da niz *prime* može biti smešten u memoriji.

U narednoj tabeli je sumirana vremenska kompleksnost svih algoritama koje smo obradili za određivanje prostih brojeva do n (Slika-2).

Kompleksnost	*Opis*
$O(n^2)$	Iscrpna pretraga, provera svih delilaca
$O(n\sqrt{n})$	Provera delilaca do $\sqrt{n}$
$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Provera prostih delilaca do $\sqrt{n}$
$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Sito Eratostena

Slika 5.2 2 Poređenje algoritama za određivanje prostih brojeva

U cilju boljeg uvida u postupak Sita Eratostena pogledati sledeći video:

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ZADATAK 1

Cilj zadatka je da se odrede svi prosti brojevi do 10,000,000,000

Zadatak 1. Napisati program koji pronalazi sve proste brojeve do **10,000,000,000**. Postoji otprilike **455,052,511** u tom intervalu. Program treba da zadovolji sledeće zahteve:

- Program treba da snimi sve proste brojeve u binarni fajl pod nazivom **PrimeNumbers.dat**. Kada se pronađe novi prost broj, onda on treba biti dodat (append) u fajl.
- Da bi odredili da li je novo-ispitani broj prost, program treba da učita proste brojeve iz fajla u niz tipa ***long***veličine maksimum **10000** elemenata. Ukoliko nijedan od elemenata niza nije delilac novog broja, onda nastaviti sa učitavanjem sledećih **10000** prostih brojeva iz fajla, sve dok se ne pronađe delilac broja ili dok se ne učitaju svi prosti brojevi iz fajla. Ukoliko nije pronađen delilac, onda je novi broj prost.

```
import java.io.*;

public class Zadatak1 {
  final static int ARRAY_SIZE = 100000;

  public static void main(String[] args) throws Exception {
    // A list to hold prime numbers
    final long N = 100000002; // Find primes up to N
    long[] primeNumbers = new long[ARRAY_SIZE];

    long number; // A number to be tested for primeness
    RandomAccessFile inout =
```

```
  new RandomAccessFile("PrimeNumbers.dat", "rw");
  if (inout.length() == 0) {
  number = 1;
  }
  else {
  inout.seek(inout.length() - 8); // A long is 8 bytes
  number = inout.readLong(); // Get the last prime number in the file
  }

  int squareRoot = 1;

  // Repeatedly find prime numbers
  newNumber:while (number <= N) {
  // Check if 2, 3, 4, ..., N is prime
  number++;
  inout.seek(0);

  if (squareRoot * squareRoot < number) {
      squareRoot++;
  }

  while (inout.getFilePointer() < inout.length()) {
      int size = readNextBatch(primeNumbers, inout);

      // if number is prime
      for (int k = 0; k < size &amp;&amp; primeNumbers[k] <= squareRoot; k++) {
      if (number % primeNumbers[k] == 0) { // If true, not prime
          continue newNumber; // Exit the for loop
      }
      }
  }

  // Append a new prime number to the end of the file
  inout.seek(inout.length());
  inout.writeLong(number);
  }

  inout.close();
}

public static int readNextBatch(long[] primeNumbers, RandomAccessFile inout) {
  int size = 0;
  try {
  while (inout.getFilePointer() < inout.length() &amp;&amp; size < ARRAY_SIZE) {
      primeNumbers[size++] = inout.readLong();
  }
  }
  catch (Exception ex) {
  ex.printStackTrace();
  }

  return size;
```

```
  }
}
```

ZADATAK 2

Cilj zadatka je da se napiše program koji će da odredi vreme izvršavanja programa za proste brojeve

Zadatak 2. Napisati program koji pronalazi vreme izvršavanja programa za određivanje prostih brojeva manjih od 8,000,000, 10,000,000, 12,000,000, 14,000,000, 16,000,000, i 18,000,000 korišćenjem algoritama obrađenih na predavanjima. Vaš program treba da oštampa sledeću tabelu:

	8000000	10000000	12000000	14000000	16000000	18000000
Provera delilaca do $\sqrt{n}$						
Provera prostih delilaca do $\sqrt{n}$						
Sito Eratostena						

Slika 5.3.1 Izgled tabele koju treba popuniti u toku izvršavanja programa

```
public class Zadatak2 {
  public static void main(String[] args) {
    long[] executionTime = new long[6];
    for (int i = 0; i < 6; i++) {
    long startTime = System.currentTimeMillis();
    findPrimeNumbers1(8000000 + i * 2000000);
    executionTime[i] = System.currentTimeMillis() - startTime;
    }
    System.out.println(
    "\t\t\t8000000\t10000000\t12000000\t14000000\t16000000\t18000000");
    System.out.println(
    "-----------------------------------------------------------------");
    System.out.print("Listing 21.4");

    for (int i = 0; i < 6; i++) {
    System.out.print("\t" + executionTime[i]);
    }

    System.out.print("\nListing 21.5");
    for (int i = 0; i < 6; i++) {
    long startTime = System.currentTimeMillis();
    findPrimeNumbers2(8000000 + i * 2000000);
    executionTime[i] = System.currentTimeMillis() - startTime;
    }

    for (int i = 0; i < 6; i++) {
    System.out.print("\t" + executionTime[i]);
    }
  }
```

```
  public static void findPrimeNumbers1(int N) {
    // final int N = 100000; // Find primes <= N
    final int NUMBER_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness

//  System.out.println("The prime numbers are \n");

    // Repeatedly find prime numbers
    while (number < N) {
    // Assume the number is prime
    boolean isPrime = true; // Is the current number prime?

    // if number is prime
    for (int divisor = 2; divisor <= (int) (Math.sqrt(number)); divisor++) {
        if (number % divisor == 0) { // If true, number is not prime
        isPrime = false; // Set isPrime to false
        break; // Exit the for loop
        }
    }

    // Print the prime number and increase the count
//      if (isPrime) {
//      count++; // Increase the count
//
//      if (count % NUMBER_PER_LINE == 0) {
//          // Print the number and advance to the new line
//          System.out.println(number);
//      }
//      else {
//          System.out.print(number + " ");
//      }
//      }

    // Check if the next number is prime
    number++;
    }
  }

  public static void findPrimeNumbers2(int N) {
    // A list to hold prime numbers
    java.util.List<Integer> list = new java.util.ArrayList<Integer>();
    // final int N = 10000000; // Find primes <= N
    final int NUMBER_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness

//  System.out.println("The prime numbers are \n");

    // Repeatedly find prime numbers
    while (number < N) {
    // Assume the number is prime
    boolean isPrime = true; // Is the current number prime?
```

```
    // if number is prime
    for (int k = 0; k < list.size() &amp;amp;&amp;amp;
                list.get(k) <= (int) (Math.sqrt(number)); k++) {
        if (number % list.get(k) == 0) { // If true, number is not prime
        isPrime = false; // Set isPrime to false
        break; // Exit the for loop
        }
    }

    // Check if the next number is prime
    number++;
    }
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Napisati program koji će naći broj prostih brojeva manjih od 10, 100, 1,000, 10,000, 100,000, 1,000,000, 10,000,000, 100,000,000, 1,000,000,000, and 10,000,000,000. Program treba da učita podatke iz tekstualnog fajla.

Zadatak 2. Napisati efikasan program za učitavanje poslednjih 100 brojeva iz fajla. (Ne treba učitavati sve brojeve redom, treba preskočiti sve do poslednjih 100).

Zadatak 3. Napisati program koji će prikazati vreme izvršavanja algoritma za pronalaženje svih prostih brojeva manjih od 8,000,000, 10,000,000, 12,000,000, 14,000,000, 16,000,000, and 18,000,000.

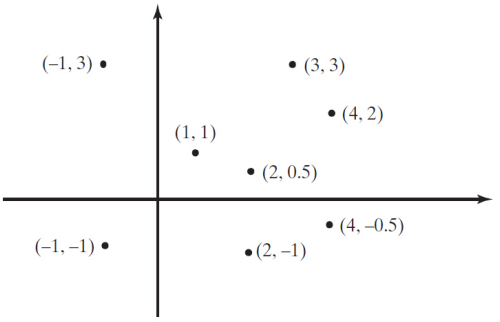
Poglavlje 6

Najbliži par tačaka – Primena algoritma „podeli i osvoji“

ALGORITAM ISCPRNE PRETRAGE

Prvo rešenje koje svima pada na pamet jeste da se izračunaju rastojanja između svih parova tačaka i da se izdvoji ono rastojanje koje je najmanje (iscrpna pretraga)

Za zadat skup tačaka u prostoru, problem najbližeg para je ustvari pronalaženje dve tačke čije je međusobno rastojanje najmanje među svim ostalim rastojanjima. Na sledećoj slici (Slika-1), n a primer, tačke **(1, 1)** i **(2, 0.5)** su najbliže jedna drugoj, računajući sve ostale tačke u skupu.

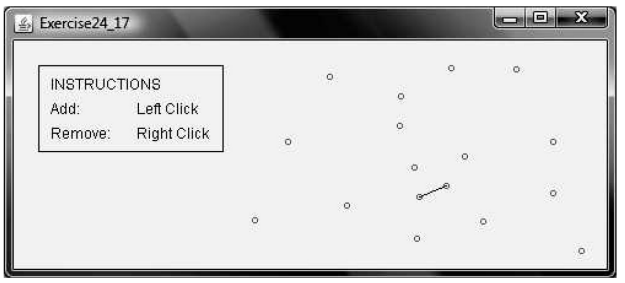


	x	y
0	–1	3
1	–1	–1
2	1	1
3	2	0.5
4	2	–1
5	3	3
6	4	2
7	4	–0.5

Slika 6.1 Tačke mogu biti predstavljene u 2D prostoru korišćenjem dvodimenzionalnog niza.

Postoji nekoliko različitih načina da se reši ovaj problem. Prvo rešenje koje svima pada na pamet jeste da se izračunaju rastojanja između svih parova tačaka i da se izdvoji ono rastojanje koje je najmanje. Ovakvo rešenje je predstavljeno u kodu koji sledi.

Kao što se može videti (Slike-2), nacrtana je linija sa ciljem da se povežu dve tačke koje se nalaze na najkraćem međusobnom rastojanju.



Slika 6.2 Animacija najbližeg para tačaka iscrtava liniju koja spaja najbliži par tačaka, i to radi dinamički, tj. kako se neka tačka interaktivno dodaje u polje ili briše iz polja tačaka.

U prethodnoj lekciji pokazan je primer određivanja najbližeg para tačaka korišćenjem algortima iscrpne pretrage. Taj algoritam je izračunavao rastojanja između svaka dva para tačaka i odredio ono sa najmanjim rastojanjem. Jasno je da je vremenska složenost takvog algoritma $O(n^2)$.

IMPLEMENTACIJA ALGORITMA ISCRPNE PRETRAGE

Program pretpostavlja da ravan ima bar dve tačke. Korisnik može lako da modifikuje program sa ciljem da se savlada i slučaj kada je uneta jedna ili nijedna tačka

```
import java.util.Scanner;

public class FindNearestPoints {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter the number of points: ");
    int numberOfPoints = input.nextInt();

    // Create an array to store points
    double[][] points = new double[numberOfPoints][2];
    System.out.print("Enter " + numberOfPoints + " points: ");
    for (int i = 0; i < points.length; i++) {
    points[i][0] = input.nextDouble();
    points[i][1] = input.nextDouble();
    }

    // p1 and p2 are the indices in the points array
    int p1 = 0, p2 = 1; // Initial two points
    double shortestDistance = distance(points[p1][0], points[p1][1],
    points[p2][0], points[p2][1]); // Initialize shortestDistance

    // Compute distance for every two points
    for (int i = 0; i < points.length; i++) {
    for (int j = i + 1; j < points.length; j++) {
        double distance = distance(points[i][0], points[i][1],
        points[j][0], points[j][1]); // Find distance

        if (shortestDistance > distance) {
        p1 = i; // Update p1
        p2 = j; // Update p2
        shortestDistance = distance; // Update shortestDistance
        }
    }
    }
```

```
    // Display result
    System.out.println("The closest two points are " +
    "(" + points[p1][0] + ", " + points[p1][1] + ") and (" +
    points[p2][0] + ", " + points[p2][1] + ")");
  }

  /** Compute the distance between two points (x1, y1) and (x2, y2)*/
  public static double distance(
    double x1, double y1, double x2, double y2) {
    return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
  }
}
```

Mogući izlaz programa bi bio:

```
Enter the number of points: 8 (enter)
Enter 8 points: -1 3   -1 -1   1 1   2 0.5   2 -1   3 3   4 2   4 -0.5 (enter)
The closest two points are (1, 1) and (2, 0.5)
```

Prethodni program zahteva od korisnika da unese broj tačaka. Tačke se učitavaju iz konzole i pakuju u dvodimenzionalni niz pod imenom *points*. Program koristi promenljivu *shortestDistance* da bi identifikovao najmanje rastojanje između dve tačke, kao i promenljive *p1* i *p2* kako bi čuvao indekse najbližih tačaka. Za svaku tačku sa indeksom *i,* program određuje rastojanje između tačaka *points[i]* i *points[j]* za svako *j > i*. Kad god je pronađeno manje rastojanje od onog koje je trenutno sačuvano u *shortestDistance*, vrši se ažuriranje promenljivih *shortestDistance*, *p1 i p2*. Rastojanje između dve tačke ***(x1, y1)*** i ***(x2, y2)*** može biti izračunato korišćenjem formule (Slika-1):

$$dist = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Program pretpostavlja da ravan ima bar dve tačke. Korisnik može lako da modifikuje program sa ciljem da se savlada i slučaj kada je uneta jedna ili nijedna tačka.

DA LI JE MOGUĆE DIZAJNIRATI EFIKASNIJI ALGORITAM?

Da. Moguće je koristiti „podeli i osvoji“ algoritam za rešenje problema. Za razliku od dinamičkog programiranja, potproblemi pristupa „podeli i osvoji“ se ne preklapaju

Koristićemo pristup "*podeli i osvoji*" da bi rešili problem. Ovaj pristup deli problem na potprobleme, rešava potprobleme, zatim kombinuje rešenja podproblema sa ciljem da se dobije rešenje kompletnog sistema. Koraci u rešavanju problema pomoću metode "*podeli i osvoji*" su.

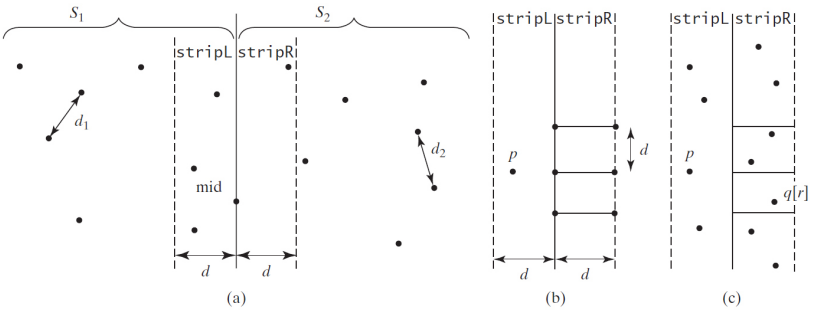
Korak 1: Sortirati tačke u rastućem poretku prema x koordinati. Za tačke koje se nalaze na istoj x koordinati sortirati po y koordinati. Ovo će dovesti do sortirane liste tačaka u prostoru. Označimo sortiranu listu sa S.

Korak 2: Podelimo sortiranu listu S u dva podskupa, S_1 i S_2, koji su jednakih veličina, korišćenjem središne tačke u sortiranoj listi. Neka je središnj tačka upala u skup S_1. Rekurzivno odrediti najbliži par u S_1 i S_2. Označimo sa d_1 i d_2 rastojanja najbližih tačaka u prvom i drugom podskupu, respektivno.

Korak 3: Odrediti najbliže rastojanje između tačaka u skupu S_1 i S_2, i označite njihovo rastojanje sa d_3. Najbliži par je jedan koji je najmanji medju vrednostima d_1, d_2, d_3, tj. min(d_1, d_2, d_3).

Korak 3 može biti izvršen za vreme *O(n)*. Neka je d = min(d_1, d_2). Mi već znamo da rastojanje najbližeg para ne može biti veće od d. Da bi za tačku u S_1 i tačku u S_2 formirali najbliži par u skupu S, mora biti zadovoljeno da je leva tačka u zoni *stripL,* a desna tačka u zoni *stripR*, kao što je i prikazano (Slika-2).

Za tačku *p* iz zone *stripL*, neophodno je razmotriti slučaj kada je desna tačka u pravougaoniku dimenzije d x 2d, kao što je prikazano na Slici-3. Bilo koja desna tačka koja se nalazi izvan ovog pravougaonika ne može formirati najbliži par zajedno sa tačkom p. Pošto je rastojanje najbližeg para veće ili jednako d, onda može postojati najviše 6 tačaka u pravougaoniku. Stoga, za svaku tačku u zoni *stripL*, najviše 6 tačaka iz zone ***stripR*** treba biti uzeto u razmatranje.



Slika 6.3 Središna tačka deli tačke skupa na dva podskupa.

ANALIZA I IMPLEMENTACIJA „PODELI I OSVOJI“ ALGORITMA

Najbliži par tačaka može biti pronađen za vreme O(n logn) kada se koristi „podeli i osvoji“ princip

Kako za svaku tačku p u zoni *stripL* locirati tačke u odgovarajućoj pravougaonoj površini d x 2d koja pripada zoni *stripR*?

Ovo može biti urađeno efikasno ako su tačke u zonama ***stripL*** i ***stripR*** sortiane u rastućem poretku prema njihovoj y koordinati. Neka sa ***pointsOrderedOnY*** označimo listu tačaka sortiranih u rastućem poretku prema y koordinati. Lista ***pointsOrderedOnY*** može biti određena unapred, pre startovanja algoritma. Zone ***stripL*** i ***stripR*** mogu biti određene iz ***pointsOrderedOnY*** u **Koraku 3**, kao što je prikazano u listingu koji sledi:

```
for each point p in pointsOrderedOnY
  if (p is in S1 and mid.x – p.x <= d)
```

```
   append p to stripL;
 else if (p is in S2 and p.x - mid.x <= d)
  append p to stripR;
```

Neka su tačke u zonama *stripL* i*stripR* označene sa {p0,p1, ... pk) i {q0,q1, ... qt} kao što je i prikazano na Slici 2c. Najbliži par između tačke iz zone *stripL* i tačke iz zone *stripR* može biti određen korišćenjem algoritma koji je opisan u narednom listingu. Kao što se može primetiti, uzeli smo u obzir da su tačke u zoni *stripL* uređene (sortirane) u poretku p_0,p_1, ... p_k za tačku p zone *stripL*, treba preskočiti tačku u zoni *stripR* koja je ispod p.y – d.

```
d = min(d1, d2);
r = 0; // r is the index in stripR
for (each point p in stripL)
{
    // Skip the points below the rectangle area
    while (r < stripR.length &amp;&amp; q[r].y <= p.y - d)
    r++;
    let r1 = r;
    while (r1 < stripR.length &amp;&amp; |q[r1].y – p.y| <= d)
    {
        // Check if (p, q[r1]) is a possible closest pair
        if (distance(p, q[r1]) < d)
        {
            d = distance(p, q[r1]);
            (p, q[r1]) is now the current closest pair;
        }

        r1 = r1 + 1;
    }
 }
```

Jednom preskočena tačka neće biti ponovo uzeta u razmatranje. Petlja ***while*** proverava da li je **(p, q[r1])** potencijalni najbliži par. Postoji najviše 6 takvih ***q[r1]*** parova, pa je složenost pronalaženja najbližeg para u *Koraku 3* reda $O(n)$. Neka je $T(n)$ vremenska složenost ovog algoritma. Imamo da je:

$T(n) = 2T(n/2) + O(n) = O(n \log n)$

Stoga, najbliži par tačaka može biti pronađen za vreme $O(n \log n)$. Kompletna implementacija ovog algoritma je ostavljena za samostalni rad.

ZADACI ZA SAMOSTALNI RAD

Na osnovu materijala uraditi samostalno sledeće zadatke

Zadatak 1. (*Najbliži par tačaka*) Na predavanjima smo se upoznali sa algoritmom za određivanje najbližeg para tačaka korišćenjem "podeli i osvoji" algoritma. Implementirati algoritam koji će da zadovolji sledeće zahteve:

- Definisati klase **Point i CompareY**.

- Definisati klasu **Pair** sa podacima **p1** i **p2** koji označavaju dve tačke, i metod pod nazivom **getDistance()** koji vraća rastojanje između dve tačke.
- Implementirati sledeće metode:

```
/** Return the distance of the closest pair of points */
public static Pair getClosestPair(double[][] points)

/** Return the distance of the closest pair of points */
public static Pair getClosestPair(Point[] points)
```

```
/** Return the distance of the closest pair of points
 * in pointsOrderedOnX[low..high]. */
public static Pair distance(Point[] pointsOrderedOnX,
int low, int high, Point[] pointsOrderedOnY)

/** Compute the distance between two points p1 and p2 */
public static double distance(Point p1, Point p2)

/** Compute the distance between points (x1, y1) and (x2, y2) */
public static double distance(double x1, double y1,
  double x2, double y2)
```

```
public static ArrayList<MyPoint> getConvexHull(double[][] s)
```

Zadatak 2. Napisati metod koji koristi strategiju "podeli i osvoji" za pronalaženje minimalnog elementa liste.

Poglavlje 7

Bektreking - Resavanje problema 8 kraljica

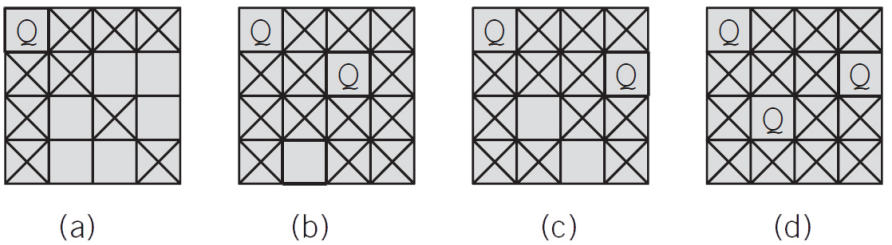
POSTAVKA PROBLEMA N KRALJICA

Problem N Kraljica se sastoji iz određivanja pozicija kraljica u svakoj od vrsta table dimenzija NxN tako da se kraljice međusobno ne napadaju

Problem N Kraljica se sastoji iz određivanja pozicija kraljica u svakoj od vrsta table dimenzija *NxN* tako da se kraljice međusobno ne napadaju. Problem se može rešiti sa ili bez rekurzije.

U okviru ove sekcije će biti opisano rešenje korišćenjem backtracking algoritma (ili algoritma vraćanja korak unazad). Ovaj algoritam inkrementalno vrši potragu za potencijalnim kandidatom rešenja, odbacujući određenu opciju čim otkrije da taj kandidat ne može biti deo tačnog rešenja, i odmah kreće u potragu za sledećim kandidatom.

Da bi smo ulustrovali problem postavljanja kraljica analiziraćemo postavljanje 4 kraljica na šahovsku tablu dimenzija *4x4*. Postupak započinjemo postavljanjem prve kraljice u prvu vrstu i prvu kolonu table (Slika-1a). Napadnuta polja su označena sa *X*. Zatim pokušavamo postavljanje druge kraljice u drugu vrstu table.



Slika 7.1 Rešavanje problema postavljanja 4 kraljice

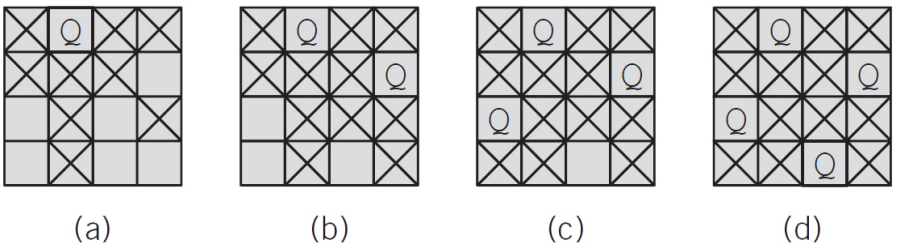
Prvo slobodno polje je u 3. koloni, pa tu i postavljamo kraljicu (Slika-1b).

Zatim pokušavamo da postavimo 3. kraljicu u 3. kolonu, ali vidimo da su sva polja napadnuta. Zato moramo da se vratimo na 2. kraljicu i pokušamo sa sledećim slobodnim poljem. To je polje u 4. koloni (Slika-1c).

Sada ponovo pokušavamo da postavimo 3. kraljicu u 3 kolonu. Jedino slovodno polje je (3,2), Slika-1d. Međutim, ne možemo dalje pošto su sva polja napadnuta.

Pošto smo obradili sve moguće slučajeve za kraljice od 2. do 3. vrste i nismo našli rešenje, moramo da se vratimo korak natrag do 1. kraljice.

Nju pomeramo na prvo sledeće polje (Slika-2a) i ponavljamo identičan postupak ispočetka. Nakon nekoliko koraka (Slika-2b i Slika-2c) dolazimo do rešenja problema (Slika-2d).



Slika 7.2 Rešenje problema postavljanja 4 kraljice

REŠAVANJE PROBLEMA OSAM KRALJICA

Problem 8 Kraljica je generalizovan problem . Problem se najlakše rešava korišćenjem bektrekinga

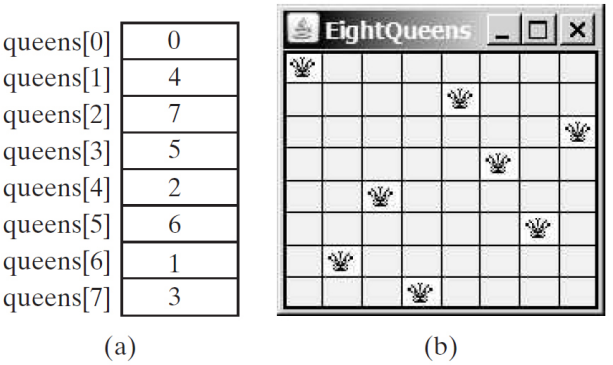
Problem 8 kraljica je specifična instanca problema postavljanja *N* kraljica.

Postupak rešavanja problema je identičan postavljanju 4 kraljice na šahovsku tablu, samo što umesto 4 sada imamo 8 kraljica.

Moguće je koristiti dvodimenzionalni niz da bi smo predstavili šahovsku tablu dimenzija 8x8. Međutim, pošto u svakoj vrsti može da se nađe samo jedna kraljica, dovoljno je da koristimo jednodimenzionalni niz da bi smo označili poziciju kraljice u toj vrsti. Stoga ćemo definisati niz ***queens*** kao:

```
int[] queens = new int[8];
```

Pritom ćemo vrednost *j* dodeliti članu *queens[i]* da bi smo označili da je kraljica postavljena u *i*-toj vrsti i *j*-toj koloni. Na Slici-3 a je prikazan sadržaj niza *queens* za šahovsku tablu sa Slike 3b. Pretraga počinje sa prvom vrstom, za koju je *k = 0*, gde je *k* indeks vrste koja se trenutno razmatra. Algoritam proverava da li kraljica može biti postavljena u *j*-toj koloni vrste, pri čemu *j* uzima vrednosti j = 0, 1, ..., 7, i to u ovom redosledu.



Slika 7.3 Vrednost člana queens[i] određuje poziciju kraljice u i-toj vrsti.

Pretraga je implementirana na sledeći način:

- Ukoliko se uspešno postavi kraljica u *k*-toj vrsti, algoritam nastavlja sa postavljanjem sledeće kraljice u sledećoj tj. *k+1*-oj vrsti. Ukoliko je trenutna vrsta poslednja (*k==8*), pronađeno je rešenje i program prekida rad.
- Ukoliko ne uspe da postavi kraljicu u tekućoj (*k*-toj) vrsti, algoritam se vraća (eng. backtrack) korak unazad na prethodnu vrstu, tj. *k-1*, i traži sledeću kolonu vrste *k-1* u kojoj može da postavi kraljicu.
- Ukoliko se algoritam vrati sve do prve vrste i obradi sve slučajeve j = 0, 1, ..., 7 u prvoj vrsti bez pronalaženja rešenja, algoritam se prekida jer nije uspeo da reši problem.

IMPLEMENTACIJA I ANALIZA BEKTREKING ALGORITMA ZA PROBLEM 8 KRALJICA

Ovaj algoritam inkrementalno vrši potragu sa potencijalnim kandidatom, odbacujući određenu opciju čim otkrije da taj kandidat ne može biti deo rešenja, i odmah kreće u potragu za sledeći

U nastavku je program koji prikazuje jedno rešenje postavljanja 8 kraljica:

```
public class EightQeens{
    public static final int SIZE = 8;
    // queens are placed at (i, queens[i])
    // -1 indicates that no queen is currently placed in the ith row
    // Initially, place a queen at (0, 0) in the 0th row
    private int[] queens = {-1, -1, -1, -1, -1, -1, -1, -1};

    /** Search for a solution */
    private boolean search() {
        // k - 1 indicates the number of queens placed so far
        // We are looking for a position in the kth row to place a queen
        int k = 0;
        while (k >= 0 &amp;amp;&amp;amp; k < SIZE) {
            // Find a position to place a queen in the kth row
            int j = findPosition(k);
            if (j < 0) {
                queens[k] = -1;
                k--; // back track to the previous row
            } else {
                queens[k] = j;
                k++;
            }
        }

        if (k == -1)
            return false; // No solution
        else
            return true; // A solution is found
    }

    public int findPosition(int k) {
```

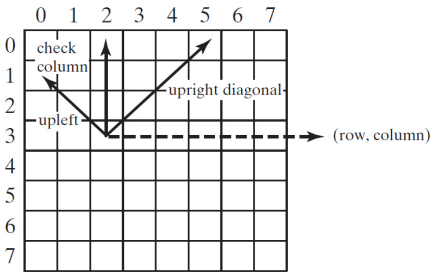
```
        int start = queens[k] + 1; // Search for a new placement

        for (int j = start; j < SIZE; j++) {
            if (isValid(k, j))
                return j; // (k, j) is the place to put the queen now
        }

        return -1;
    }

    /** Return true if a queen can be placed at (row, column) */
    public boolean isValid(int row, int column) {
        for (int i = 1; i <= row; i++)
            if (queens[row - i] == column // Check column
            || queens[row - i] == column - i // Check upleft diagonal
            || queens[row - i] == column + i) // Check upright diagonal
                return false; // There is a conflict
        return true; // No conflict
    }
}
```

Kod: http://www.cs.armstrong.edu/liang/intro9e/html/EightQueens.html.

Metod *search()* traži rešenje problema. Pretraga počinje sa prvom vrstom tj. sa *k = 0*. Ukoliko je pretraga uspešna postavljamo kraljicu na to mesto i prelazimo u sledeću vrstu. Ukoliko ne uspemo da nađemo mesto u *k*-toj vrsti, vraćamo se (eng. backtrack) na prethodnu vrstu.

Metod *findPosition(k)* traži moguću poziciju da postavi kraljicu u *k*-toj vrsti počevši od pozicije *queen[k] + 1*. On proverava da li nova kraljica može biti postavljena na pozicijama *start*, *start + 1*,..., i *7*. Ako je moguće, vratiti indeks kolone u kojoj smo postavili kraljicu; u suprotnom vratiti *-1*.

Metod *isValid(row, column) proverava da li moguće postavljanje nove kraljice na poziciju (row, column)*, tj. da li je ta pozicija napadnuta od prethodno postavljenih kraljica (Slika-4).

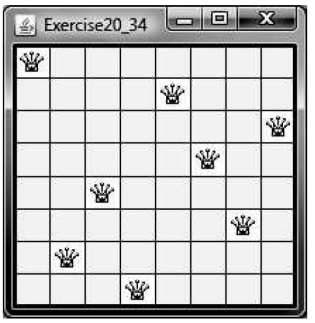


Slika 7.4 Poziv metode isValid(row, column) koja ispituje da li nova kraljica može biti postavljena na poziciju (row, column)

ZADATAK 1: REKURZIVNO REŠENJE

Cilj zadatka je implementacija rekurzivnog bektreking algoritma za problem 8 kraljica u programskom jeziku Java

Napisati program u Java-i koji rekurzivno rešava Problem 8 Kraljica, i rezultat prikazuje u obliku kao na Slici- 1 .



Slika 7.5 Program koji prikazuje rešenje za problem 8 kraljica

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

public class Exercise18_34 extends Application {
  private static final int SIZE = 8;
  private int[] queens = new int[SIZE]; // Queen positions

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    search(0); // Search for a solution from row 0

    ChessBoard board = new ChessBoard();

    // Create a scene and place it in the stage
    Scene scene = new Scene(board, 250, 250);
    primaryStage.setTitle("Exercise18_34"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    board.paint();

    scene.widthProperty().addListener(ov -> board.paint());
    scene.heightProperty().addListener(ov -> board.paint());
  }

  /** Check if a queen can be placed at row i and column j */
  private boolean isValid(int row, int column) {
    for (int i = 1; i <= row; i++)
      if (queens[row - i] == column // Check column
        || queens[row - i] == column - i // Check upleft diagonal
        || queens[row - i] == column + i) // Check upright diagonal
        return false; // There is a conflict
    return true; // No conflict
```

```
  }

  /** Search for a solution starting from a specified row */
  private boolean search(int row) {
    if (row == SIZE) // Stopping condition
      return true; // A solution found to place 8 queens in 8 rows

    for (int column = 0; column < SIZE; column++) {
      queens[row] = column; // Place a queen at (row, column)
      if (isValid(row, column) &amp;amp;&amp;amp; search(row + 1))
        return true; // Found, thus return true to exit for loop
    }

    // No solution for a queen placed at any column of this row
    return false;
  }

  private class ChessBoard extends Pane {
    Image queenImage = new Image("image/queen.jpg");

    public void paint() {
      // Clear previous drawing
      this.getChildren().clear();

      // Draw the queens
      for (int i = 0; i < SIZE; i++) {
        // Add the queen image view
        ImageView queenImageView = new ImageView(queenImage);
        this.getChildren().add(queenImageView);
        int j = queens[i]; // The position of the queen in row i
        queenImageView.setX(j * getWidth() / SIZE);
        queenImageView.setY(i * getHeight() / SIZE);
        queenImageView.setFitWidth(getWidth() / SIZE);
        queenImageView.setFitHeight(getHeight() / SIZE);
      }

      // Draw the lines
      for (int i = 1; i <= SIZE; i++) {
        this.getChildren().add(
          new Line(0, i * getHeight() / SIZE, getWidth(), i * getHeight() / SIZE));
        this.getChildren().add(
          new Line(i * getWidth() / SIZE, 0, i * getWidth() / SIZE, getHeight()));
      }
    }
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    launch(args);
```

```
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Napisati program koji pronalazi sva moguća rešenja problema *osam kraljica*. Označiti svako rešenje rednim brojem.

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 zateva 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.

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, deseto izdanje, Prentice Hall, 2014

[2] M.A. Weiss, Data Structures and Problem Solving Using Java, treće izdanje, Addison Wesley, 2005.

[3] M.A. Weiss, Data Structures and Algorithm Analysis in Java, treće izdanje, Pearson Education, 2012.

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

[5] https://en.wikipedia.org/wiki/Analysis_of_algorithms

[6] https://en.wikipedia.org/wiki/Big_O_notation