UNIVERZITET METROPOLITAN BEOGRAD

CS203 - ALGORITMI I STRUKTURE PODATAKA

Razvoj efikasnog algoritma

Lekcija 03

PRIRUČNIK ZA STUDENTE

CS203 - ALGORITMI I STRUKTURE PODATAKA

Lekcija 03

RAZVOJ EFIKASNOG ALGORITMA

- Razvoj efikasnog algoritma
- Poglavlje 1: Pretraživanje nizova
- Poglavlje 2: Efikasni algoritmi za određivanje prostih brojeva
- Poglavlje 3: Maksimalna suma podsekvence niza
- Poglavlje 4: Bektreking algoritam: Problem 8 kraljica
- Poglavlje 5: Pokazne vežbe
- Poglavlje 6: Zadaci za samostalan rad
- Poglavlje 7: Domaći zadatak
- Zaključak

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upoznaju sa primerima razvoja efikasnog algoritma:

- Linearno i binarno pretraživanje, i vremenska analiza oba pristupa
- Razvoj efikasnih algoritama za određivanje prostih brojeva.
- Analiza i implementacija efikasnih numeričkih i geometrijskih algoritama.
- Problem 8 kraljica - primena bektreking algoritma

Poglavlje 1

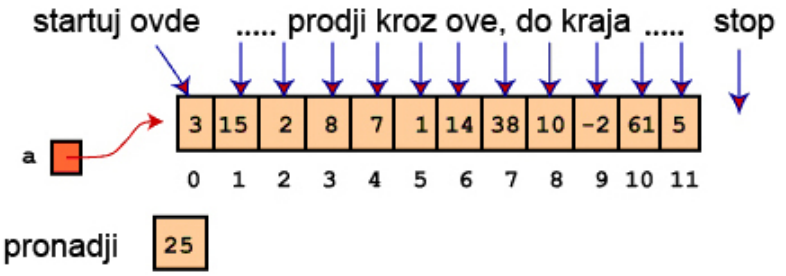
Pretraživanje nizova

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. Dve najpoznatije metode su ***linearno*** i ***binarno pretraživanje*** .

Pristup linearnog pretraživanja upoređuje ključ pretrage *key* 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 1.1). Ukoliko se ne pronađe ključ u nizu, algoritam vraća vrednost *-1*. Metod linearnog pretraživanja je prikazan u sledećem listingu.



Slika 1.1 Primer linearnog pretraživanja niza [1]

Implementacija lineranog pretraživanja je data u nastavku:

```
public class LinearSearch {
    /** Metod za pronalazenje kljuca u listi */
    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);  // Vraca 1
int j = linearSearch(list, -4); // Vraca -1
int k = linearSearch(list, -3); // Vraca 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.

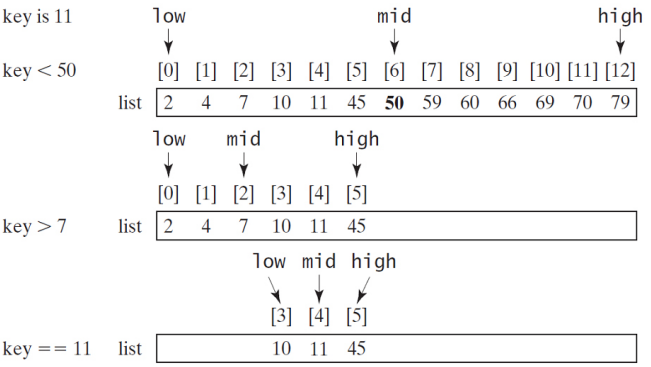
Pošto vreme izvršavanja linearnog pretraživanja raste linearno sa brojem elemenata niza, možemo zaključiti da je linearno pretraživanje neefikasan postupak za velike nizove. Za naredni metod, binarno pretraživanje nam treba niz koji je već sortiran (strategija transform and conquer).

BINARNO PRETRAŽIVANJE

Da bi binarno pretraživanje moglo da se primeni, niz vrednosti mora biti sortiran (uređen). Pritom postoje tri slučaja koja se ispituju

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.



Slika 1.2 Binarno pretraživanje eliminiše polovinu niza u svakom sledećem koraku pretrage [1]

Na Slici 1.2 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.

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

Pošto sada znate kako radi binarno pretraživanje, ostalo je da ga implementirate u nekom programskom jeziku.

ALGORITAM BINARNOG PRETRAŽIVANJA

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

Algoritam i kod za binarno pretraživanje se može projektovati inkrementalno korak po korak.

Možete započeti prvu iteraciju pretrage kao što je prikazano na Slici 1.3a. 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*** ;

```
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 1.3 Inkrementalni algoritam binarne pretrage [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 implementaciju metoda dodavanjem petlje, u okviru koje će da se obavi kompletna pretraga, kao što je prikazano na Slici 1.3b. Pretraga se prekida kada je ključ pronađen, ili kada obavimo sve korake petlje pretrage, tj. kada je *low > high*.

U slučaju da ključ nije pronađen, indeks *low* postaje tačka insertovanja gde ključ treba da bude 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

U toku svake iteracije algoritma vrši se pomeranje granica low ili high što utiče da se u narednom ciklusu podniz koji se ispituje skrati za polovinu

Šta da uradimo da bismo 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. Vraćanjem vrednosti ***–low – 1*** označavamo ne samo da ključ nije pronađen u listi, već i mesto gde ključ treba da bude 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

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

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 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.

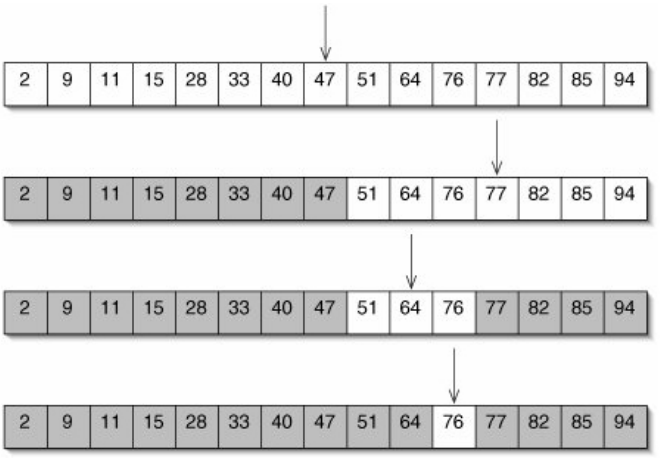
Animaciju rada algoritma možete pogledati na linku:

https://liveexample.pearsoncmg.com/liang/animation/web/BinarySearchNew.html

VREMENSKA ANALIZA BINARNOG PRETRAŽIVANJA

Pošto binarno pretraživanje eliminiše polovinu niza nakon svake iteracije pretrage, njegova složenost 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 1.4).



Slika 1.4 Postupak binarnog pretraživanja - pretraga broja 76 u nizu brojeva [izvor: autor]

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 1.5):

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

Slika 1.5 Vremenska složenost algoritma binarnog pretraživanja [1]

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

Jedan algoritam koji ima vremensku složenost O(logn) se naravno naziva logaritamski algoritam. Prirodu logaritamskih algoritama smo opisali u prethodnoj sekciji.

Poglavlje 2

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

U prethodnoj lekciji 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(√n)*.

Program neće biti efikasan ako morate da određujete koren, tj. da pozivate *Math.sqrt(number)* u svakoj iteraciji for petlje. Zato je bolje da se *Math.sqrt(number)* pozove samo jednom, i to pre ulaska u for petlju.

Pretpostavimo da želimo da odredimo prvih *n* prostih brojeva. U tom slučaju bi za svaki broj koga proveravamo trebalo da odredimo njegov koren. Kod može biti napisan na sledeći način:

```
int number = 2;
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++; // Increase the count
        System.out.printf("%7d\n", number);
    }
```

```
    number++;
}
```

Medjutim, nema potrebe da *Math.sqrt(number)* bude sračunato za svaki broj. Naime, dovoljno je da proverimo brojeve čiji je koren ceo broj, tj 4, 9, 16, 25, 36, 49, itd. Primetimo da je za brojeve od 36 do 48, njihov*(int)(Math.sqrt(number))*jednak 6. Imajući ovo u vidu možemo da izmenimo početni deo while petlje i da dobijemo još efikasniji algoritam:

```
int number = 2;
int squareRoot = 1;
while (number <= n) {
    boolean isPrime = true;
    if (squareRoot * squareRoot < number) squareRoot++;
    for (int divisor = 2; divisor <= squareRoot; divisor++) {
        ...
    }
}
```

Vremenska složenost ovakvog algoritma će biti O(n√n).

PRIMENA 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 √i. Međutim, dovoljno je samo da proverimo da li je i deljivo prostim brojevima između 2 i √i. 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 $p \leq q$ (dokaz je izvan opsega ovog kursa). Evo dokaza. Pretpostavimo da *i* nije prost broj: neka je *p* najmanji faktor broja *i, p* mora biti prost jer su suprotnom, *p* će biti deljivo sa *k*, pri čemu je $2 \leq k < p$. *k* je takođe delilac broja *i*, što je kontradiktorno sa pretpostavkom da je *p* najmanji delilac broja *i*.

Ovo vodi do još efikasnijeg algoritma koji određuje sve proste brojeve do *n*. Ovaj algoritam je još jedan od primera dinamičkog programiranja. Algoritam čuva rezultate potproblema 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 EfficientPrimeNumbers {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Odredi sve proste brojeve <= n, unesi n: ");
    int n = input.nextInt();

    // Lista koja cuva proste brojeve
    java.util.List<Integer> list =
```

```
      new java.util.ArrayList<Integer>();

    final int NUMBER_PER_LINE = 10; // Prikazi 10 u jednoj liniji
    int count = 0; // Brojac prostih brojeva
    int number = 2; // Broj koga proveravamo da li je prost
    int squareRoot = 1; // Proveri da li je number <= squareRoot

    System.out.println("Prosti brojevi su \n");

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

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

      // Proveri da li je broj prost
      for (int k = 0; k < list.size()
                       && list.get(k) <= squareRoot; k++) {
        if (number % list.get(k) == 0) { // Ako je tacno, number nije prost
          isPrime = false; // Postavi isPrime na false
          break; // Izadji iz for petlje
        }
      }

      if (isPrime) {
        count++; // Uvecaj brojac prostih
        list.add(number); // Dodaj novo-otkriveni prost broj u listu
        if (count % NUMBER_PER_LINE == 0) {
          // Odstampaj broj i predji u sledecu liniju
          System.out.println(number);
        }
        else
          System.out.print(number + " ");
      }
      number++;
    }

    System.out.println("\n" + count +
      " prostih brojeva manjih ili jednakih" + n);
  }
}
```

ANALIZA ALGORITMA DINAMIČKOG PROGRAMIRANJA

Vremenska složenost algoritma koji koristi dinamičko programiranje je O(n√n/logn).

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 √i; Broj prostih brojeva manjih ili jednakih √i je:

√i / log √i = 2√i / log i.

Tako, vremenska složenost određivanja svih prostih brojeva do n je:

2√2 / log 2 + 2√3 / log 3 + 2√4 / log 4 + ... + 2√n / log n.

Pošto je √i / log i < √n / log n, za i < n i n >= 16, imamo da je:

2√2 / log 2 + 2√3 / log 3 + 2√4 / log 4 + ... + 2√n / log n < 2n√n / log n.

Stoga možemo zaključiti da je složenost ovog algoritma O(n√n/logn).

Da li postoji bolji algoritam od O(n√n/logn)? Naravno! Analizirajmo sada Eratostenov algoritam za odredjivanje prostih brojeva.

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 kreirao veoma pametan algoritam, poznat kao ***Sieve of Eratosthenes*** , za određivanje svih prostih brojeva ≤ n. Algoritam uzima 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 ≤ *i* ≤ *n*/2, kao što je prikazano na Slici 2.1. Pošto su za nas zanemarive vrednosti 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 2.1 Vrednosti u nizu primes se menjaju nakon obrade svakog sledećeg broja k [1]

Pošto svi brojevi deljivi sa 3 nisu prosti, postaviti članove *primes[3 * i]* na vrednost false za svako 3 ≤ i ≤ n/3. Kako svi brojevi pomnoženi sa 5 nisu prosti, postaviti takođe *primes[5 * i]* na false za sve 5 ≤ *i* ≤ *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. Napomenimo da je k <= n / k. U suprotnom, k * i bi bilo veće od n.

```
import java.util.Scanner;
public class SieveOfEratosthenes {
 public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  System.out.print("Odredi sve proste brojeve <= n, unesi n: ");
  int n = input.nextInt();

  boolean[] primes = new boolean[n + 1]; // Niz indikatora prostih brojeva

  // Pocetno, postavi primes[i] na 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 nije prost
    }
   }
  }

  int count = 0; // Broj prostih koji je do sada pronadjen
  // Odstampaj prost broj
  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 +
    " prostih brojeva manjih ili jednakih " + n);
 }
}
```

ANALIZA ALGORITMA SITA ERATOSTENA

Složenost 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 složenost svih algoritama koje smo obradili za određivanje prostih brojeva do n (Slika 2.2).

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 2.2 Poređenje algoritama za određivanje prostih brojeva [1]

U cilju boljeg uvida u postupak Eratostenobog sita pogledati sledeći video:

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

Poglavlje 3

Maksimalna suma podsekvence niza

OPIS PROBLEMA I KUBNI ALGORITAM

Problem se sastoji u određivanju najvećeg neprekidnog podniza tako da je suma elemenata podniza najveća. Najprostiji algoritam se svodi na direktnu iscrpnu pretragu (kubni algoritam)

Jedna bitna tehnika za rešavanje problema koja koristi rekurziju je **"*podeli i osvoji*"** . U ovoj sekciji ćemo uvesti problem odredjivanja maksimalne podsekvence niza i prikazati kako se on može efikasno rešiti primenom ove. Za početak analizirajmo rešenja primenom iscrpne pretrage.

Ako je dat niz celih brojeva A_1, A_2, ... A_n treba pronaći podniz k, j (kao i indekse k i j) čija je suma elemenata:

$$\sum_{k=1}^{j} A_k$$

maksimalna. U slučaju da su svi elementi niza negativni usvojiti da je maksimalna suma jednaka nuli.

Na primer, ako je ulazni niz {–2, **11**, **–4**, **13**, –5, 2} onda je odgovor *maxSuma = 20*, a indeksi idu od 2 do 4. U slucaju da je ulazni niz { 1, –3, **4**, **–2**, **–1**, **6** } odgovor treba da bude *maxSuma = 7*, a indeksi idu od 3 do 6.

U okviru ove sekcije biće opisano nekoliko različitih algoritama.Najprostiji algoritam je algoritam iscrpne pretrage:

Za svaku mogući podniz izračunati njegovu sumu. Ukoliko je to trenutno najveća izračunata suma onda njenu vrednost sačuvati u *maxSuma*.

Takođe ažuriramo i dve promenljive *nStart* i *nKraj* - koje čuvaju početni I krajnji indeks podsekvence kada se odredi trenutno najbolja suma.

```
public static int maxSuma(int [] Niz, int nStart, int nKraj){
    int maxSuma = 0;

    for( int i = 0; i < Niz.length; i++ ) {
        for( int j = i; j < Niz.length; j++ ) {
            int tempSuma = 0;
            for( int k = i; k <= j; k++ )
```

```
                tempSuma += Niz[k];

            if( tempSuma > maxSuma ){
                maxSuma = tempSuma;
                nStart  = i;
                nKraj   = j;
            }
        }
    }

    return maxSuma;
}
```

Algoritam se sastoji iz tri ugneždene ***for*** petlje pri čemu se svaka od njih izvršava N puta. Složenost je $O(N^3)$ jer se ispituju sve moguće kombinacije podnizova između indeksa I i J - (i,j) (i = 1,...,N; j = i,...,N). Zatim se za tu kombinaciju koristi petlja koja računa sumu elemenata.

KVADRATNI ALGORITAM

Poboljšani algoritam se sastoji u korišćenju već izračunatih suma kako dodavanjem novog člana u podiz ne bi računali sve ispočetka. Na ovaj način izbacujemo jednu petlju

Kada bismo mogli da izbacimo bar jednu ugneždenu petlju iz algoritma, generalno bismo smanjili vreme izvršavanja algoritma. Postavlja se pitanje kako je moguće da se izbaci neka unutrašnja petlja?

U našem prethodnom algoritmu postoji mnogo nepotrebnih izračunavanja. Neefikasnost koju bi unapređeni algoritam korigovao se odnosi na prekomerna i bezrazložna izračunavanja u unutrašnjoj ***for*** petlji. Poboljšani algoritam bi trebalo da uzme sledeću činjenicu u obzir:

$$\sum_{k=1}^{j} A_k = \sum_{k=1}^{j-1} A_k + A_j$$

Drugim rečima, pretpostavimo da smo upravo izračunali zbir podniza i, ..., j-1. Nakon toga, izračunavanje ukupnog zbira za podniz i, ..., j ne bi trebalo da traje dugo, jer nam je neophodna još samo jedna operacija sabiranja. Ukoliko uzmemo ovo u obzir, dobićemo poboljšani algoritam koji je predstavljen u nastavku.

Sada imamo dve umesto tri for petlje, pa je vreme izvršavanja programa $O(n^2)$.

```
public static int maxSumb ( int [] Niz, int nStart, int nKraj ) {
    int maxSumb = 0;

    for( int i = 0; i < Niz.length; i++ ) {
        int tempSuma = 0;

        for( int j = i; j < Niz.length; j++ ) {
```

```
            tempSuma += Niz[j];
            if( tempSuma > maxSumb ) {
                maxSumb = tempSuma;
                nStart  = i;
                nKraj   = j;
            }
        }
    }

    return maxSumb;
}
```

EFIKASNO REŠENJE PROBLEMA

Problem se sastoji u određivanju najvećeg neprekidnog podniza tako da je suma elemenata podniza najveća. U nastavku će biti opisani algoritmi različite složenosti

U ovoj sekciji ćemo se pozabaviti dobro poznatim problemom:

Ako je dat niz celih brojeva A_1,A_2, ... A_n treba pronaći podniz k, j (kao i indekse k i j) čija je suma elemenata:

$$\sum_{k=1}^{j} A_k$$

maksimalna. U slučaju da su svi elementi niza negativni usvojiti da je maksimalna suma jednaka nuli.

Pre nego što započnemo diskusiju o algoritmima za rešenja ovog problema, treba i da prokomentarišemo specijalan slučaj kada su svi brojevi negativni. Za ovaj slučaj, usvojićemo da je maksimalna suma jednaka 0. Postavlja se pitanje zašto ne pronađemo najveći negativni broj među njima i ne proglasimo taj broj za najveći. Razlog je taj što prazna podsekvenca je takođe kontinualna podsekvenca čija je suma jednaka nuli. S obzirom da je prazna podsekvenca kontinualna, onda uvek postoji kontinualna podsekvenca čija je suma jednaka nuli. Rezultat je analogan slučaju praznog skupa koji je podskup nekog većeg skupa. Budite svesni toga da praznost skupa je uvek mogućnost koju možete da koristite.

Problem maksimalne sume kontinualne podsekvence brojeva je veoma zanimljiv uglavnom iz razloga što postoji ogroman broj algoritama za njegovo rešavanje — a performanse ovih algoritama se drastično razlikuju.

Ranije je opisano nekoliko različitih algoritama. Prvi je očigledan algoritam iscrpne pretrage koji je veoma neefikasan. Drugi je poboljšanje, koje se dobija prostim posmatranjem i zapažanjima. Taj algoritam je imao kvadratnu složenost. U okviru lekcije o rekurziji smo se upoznali sa "podeli i osvoji" algoritmom čija je složenost bila O(n log n).

U okviru ove sekcije ćemo opisati veoma efikasan ali ne i očigledan algoritam. Dokazaćemo da je vreme njegovog izvršavanja linearno.

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

POBOLJŠANJE KVADRATNOG ALGORITMA

U cilju poboljšanja algoritma koristimo sledeću teoremu: Neka je Ai,j podniz čija je suma elemenata Si,j < 0 . Ako je q > j, onda Ai,q nije najveći kontinualni podniz

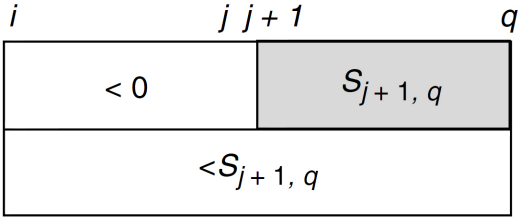
Da bi smo poboljšali algoritam sa kvadratne na linearnu kompleksnost, neophodno je da izbacimo još jednu for petlju. Međutim, za razliku od primera gde je kod poboljšanja sa kubne na kvadratnu kompleksnost izbacivanje jedne for petlje bilo prosto, u ovom slučaju izbacivanje još jedne for petlje nije baš tako jednostavno. Problem je u tome što je kvadratni algoritam i dalje iscrpna pretraga, što znači da tražimo i dalje sve moguće podnizove. Jedina razlika između kvadratnih i kubnih algoritama je u tome što je vreme testiranja svakog uzastopnog podniza konstantno $O(1)$ a ne linearno $O(n)$. Pošto imamo kvadratni broj podnizova koje istražujemo, jedini način da postignemo granicu ispod kvadratne je da pronađemo pametniji način da eliminišemo razmatranje velikog broja podnizova, ali zapravo bez određivanja njihovih suma i provera da se vidi da li je ta suma upravno novi maksimum. U nastavku ćemo opisati kako je ovo urađeno.

Prvo, potrebno je da eliminišemo ogroman broj podnizova iz razmatranja. Jasno, najbolji podniz nikako ne može počinjati negativnim brojem, pa tako ako je a[i] negativno onda možemo preskočiti celu unutrašnju for petlju i uvećati brojač ***i*** za 1. Još opštije, važi pretpostavka da najbolji podniz ne može počinjati podnizom negativnih brojeva.

Stoga, neka je $A_{i,j}$ podniz brojeva između i i j koje još nismo obradili, i neka je $S_{i,j}$ njihova suma.

Teorema 1: Neka je $A_{i,j}$ podniz čija je suma elemenata $S_{i,j} < 0$. Ako je $q > j$, onda $A_{i,q}$ nije najveći kontinualni podniz.

Jedna ilustracija suma generisanih korišćenjem indeksa *i*, *j*, i *q* je prikazana na Slici 3.1. P rethodna Teorema 1 demonstrira da mi možemo da izbegnemo razmatranje nekih podsekvenci tako što uključujemo dodatnu proveru: Ukoliko je thisSum manja od 0, možemo prekinuti (korišćenjem break) unutrašnju petlju.



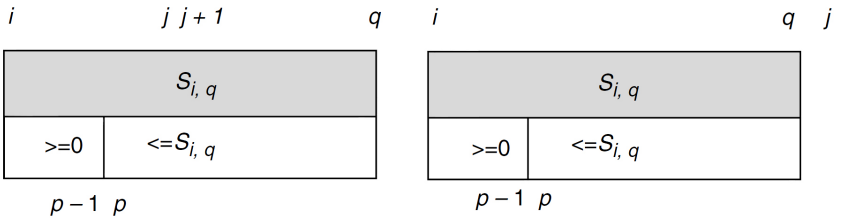
Slika 3.1 Podsekvenca korišćena u teoremi 1 [1]

Intuitivno, ukoliko je suma podsekvence negativna onda ne može biti deo najveće kontinualne podsekvence. Razlog je taj što dobijamo veću kontinualnu podsekvencu ako prethodno pomenutu negativnu ne uključimo. Ovo razmatranje samo po sebi nije dovoljno da se smanji vreme izvršavanja algoritma ispod kvadratnog.

ANALIZA I IMPLEMENTACIJA LINEARNOG ALGORITMA

Za neko i, neka je Ai,j prva sekvenca za koju je Si,j < 0. Sledi da, za neko i ≤ p ≤ j and p ≤ q, Ap,q ili nije najveća kontinualna sekvenca ili je jednaka već viđenoj najvećoj podsekvenci

Uzmimo u obzir naredno razmatranje: Sve kontinualne podsekvence koje se graniče sa najvećom kontinualnom podsekvencom moraju imati sumu manju ili jednaku nuli (u suprotnom, podsekvenca bi bila uključena u najveću). Ovo razmatranje i dalje ne smanjuje vreme izvršavanja algoritma ispod kvadratne kompleksnosti. Sledeći slučaj, međutim, poboljšava algoritam ispod kvadratne kompleksnosti. Ilustracija algoritma je prikazana na Slici 3.2. Ovaj slučaj ćemo formulisati korišćenjem teoreme:



Slika 3.2 Podsekvenca koju koristimo u teoremi 2. Sekvenca brojeva od p do q ima sumu koja je u krajnjem slučaju ista kao ona za podsekvencu od i do q. Na levoj strani sekvenca od i do q sama po sebi nije minimalna (po teoremi 1). Na desnoj strani sekvenca od i do q je već obrađena. [1]

Teorema 2: Za neko *i*, neka je $A_{i,j}$ prva sekvenca za koju je $S_{i,j} < 0$. Sledi da, za neko $i \le p \le j$ i $p \le q$, $A_{p,q}$ ili nije najveća kontinualna sekvenca ili je jednaka već viđenoj najvećoj kontinualnoj podsekvenci.

Dokaz: Ukoliko je p=i, onda primenjujemo Teoremu 1. U suprotnom, kao i u Teoremi 1, imamo da je $S_{i,q} = S_{i,p-1} + S_{p,q}$. Pošto je j najmanji indeks za koji je $S_{i,j} < 0$, sledi da je $S_{i,p-1} \ge 0$. Stoga je $S_{p,q} \le S_{i,q}$. Ako je q > j (Slika-2 levo), onda Teorema 1 podrazumeva da $A_{i,q}$ nije najveća kontinualna podsekvenca, pa nije ni $A_{p,q}$. Suprotno tome (Slika 3.2 desno), podsekvenca $A_{p,q}$ ima sumu koja je u graničnom slučaju jednaka već viđenoj (obrađenoj) podsekvenci. Očigledno je da je kompleksnost algoritma O(N).

```
public static int maxSumc ( int [ ] Niz, int nStart, int nKraj )
{
 int maxSumc  = 0;
 int tempSuma = 0;
 for( int i = 0, j = 0; j < Niz.length; j++ )
 {
```

```
  tempSuma += Niz[j];
  if( tempSuma > maxSumc )
  {
   maxSumc = tempSuma;
   nStart  = i;
   nKraj   = j;
  }
  else if( tempSuma < 0 )
  {
   i = j + 1;
   tempSuma = 0;
  }
 }
 return maxSumc;
}
```

Poglavlje 4

Bektreking algoritam: Problem 8 kraljica

REŠAVANJE PROBLEMA OSAM KRALJICA

Problem 8 Kraljica je generalizovan problem . Problem se najlakše rešava primenom obrnute pretrage

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.

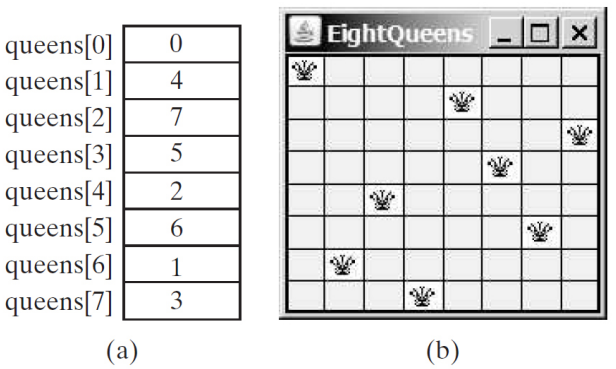
Problem 8 kraljica je specifična instanca problema postavljanja *N* kraljica.

Postupak rešavanja problema je identičan postavljanju 4 kraljice na šahovsku tablu, što smo imali u L01, 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 bismo 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 bismo označili da je kraljica postavljena u *i*-tu vrstu i *j*-tu kolonu. Na Slici 4.1(a) je prikazan sadržaj niza *queens* za šahovsku tablu sa Slike 4.1(b). 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*-tu kolonu vrste, pri čemu *j* uzima vrednosti j = 0, 1, ..., 7, i to u ovom redosledu.



Slika 4.1 Vrednost člana queens[i] određuje poziciju kraljice u i-toj vrsti. [1]

Pretraga je implementirana na sledeći način:

- Ukoliko se uspešno postavi kraljica u *k*-tu vrstu, algoritam nastavlja sa postavljanjem sledeće kraljice u sledeću tj. *k+1*-vu vrstu. Ukoliko je trenutna vrsta poslednja (*k==8*), pronađeno je rešenje i program prekida rad.
- Ukoliko ne uspe da postavi kraljicu u tekuću (*k*-tu) vrstu, algoritam se vraća (eng. backtrack) korak unazad na prethodnu vrstu, tj. *k-1*, i traži sledeću kolonu vrste *k-1* u koju 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 ALGORITMA

Problem 8 kraljica korišćenjem obrnute pretrage se veoma lako može rešiti i rekurzijom. Takav algoritam se u literaturi često naziva "rekurzivni bektreking"

U nastavku je dat JavaFX program koji rekurzivno rešava Problem 8 Kraljica, i rezultat prikazuje u obliku kao na Slici 4.2 .

Slika 4.2 Program koji prikazuje rešenje za problem 8 kraljica [1]

```
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 Recursive8Queen extends Application {
  private static final int SIZE = 8;
  private int[] queens = new int[SIZE]; // Pozicije kraljica

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    search(0); // Pronadji resenje pocev od 0-te vrste

    ChessBoard board = new ChessBoard();

    // Kreiraj prikaz (scene) i postavi je na stage (scenu)
    Scene scene = new Scene(board, 250, 250);
```

```
    primaryStage.setTitle("Recursive8Queen"); // Podesi naslov (title) scene
    primaryStage.setScene(scene); // Postavi prikaz na scenu (stage)
    primaryStage.show(); // Prikazi scenu (stage)

    board.paint();

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

  /** Proveri da li kraljica moze biti postavljena u i-tu vrstu i j-tu kolonu
  private boolean isValid(int row, int column) {
    for (int i = 1; i <= row; i++)
      if (queens[row - i] == column // Provei kolonu
        || queens[row - i] == column - i // Proveri glavnu dijagonalu
        || queens[row - i] == column + i) // Proveri sporednu dijagonalu
        return false; // Postoji napadanje sa drugom kraljicom
    return true; // Ne postoji napadanje sa drugom kraljicom
  }

  /** Potraga za resenjem pocev od navedene vrste */
  private boolean search(int row) {
    if (row == SIZE) // Uslov zaustavljanja
      return true; // Pronadjeno je resenje za postavljanje 8 kraljica u 8 vrsta

    for (int column = 0; column < SIZE; column++) {
      queens[row] = column; // Postavvi kraljicu u polje (row, column)
      if (isValid(row, column) && search(row + 1))
        return true; // Pronadjeno, vrati true i prekini for petlju
    }

    // Ne postoji resenje za postavljanje kraljice u bilo koje polje ove vrste
    return false;
  }

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

    public void paint() {
      // Obrisi prethodni prikaz scene
      this.getChildren().clear();

      // Prikazi kraljice
      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]; // Pozicija kraljice u i-toj vrsti
        queenImageView.setX(j * getWidth() / SIZE);
        queenImageView.setY(i * getHeight() / SIZE);
        queenImageView.setFitWidth(getWidth() / SIZE);
        queenImageView.setFitHeight(getHeight() / SIZE);
      }
```

```
      // Iscrtaj linije
      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()));
      }
    }
  }

  /**
   * Metod main() je potreban samo za IDE sa ogranicenom
   * JavaFX podrskom. Nije potreban za pokretanje iz komandne linije.
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

ANALIZA ALGORITMA

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

Kao sto smo videli, ***Problem 8 Kraljica*** ima za cilj postavljanje kraljica u svaku vrstu šahovske table tako da se kraljice međusobno ne napadaju (kraljice se napadaju ako se nalaze u istoj vrsti/koloni odnosno u istoj dijagonali).

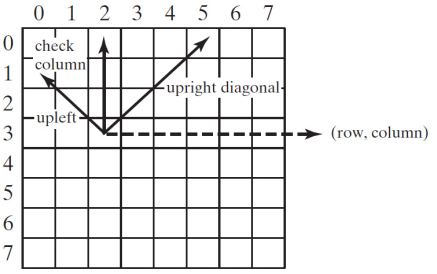
Bez korišćenja obrnute pretrage, već samo primenom grube sile, problem bi bilo složenosti $O(2^N)$. Uz primenu obrnute pretrage, može da se pokaže da je vreme izvršavanja svedeno na O(N!) :

$T(N) = O(N^2) + N*T(N-1) = O(N^2) + O(N!) = O(N!)$

U nastavku je dat JavaFX program koji rekurzivno rešava Problem 8 Kraljica, i rezultat prikazuje u obliku kao na Slici 4.3 .

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 *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 4.3).



Slika 4.3 Poziv metode isValid(row, column) koja ispituje da li nova kraljica može biti postavljena na poziciju (row, column) [1]

Poglavlje 5

Pokazne vežbe

VREME IZVRŠAVANJA PROGRAMA ZA ODREDJIVANJE PROSTIH BROJEVA (20 MIN)

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

Zadatak 1. 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. (20 min)

Vaš program treba da odštampa sledeću tabelu (Slika 5.1):

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

Slika 5.1 Izgled tabele koju treba popuniti u toku izvršavanja programa [1]

```
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);
```

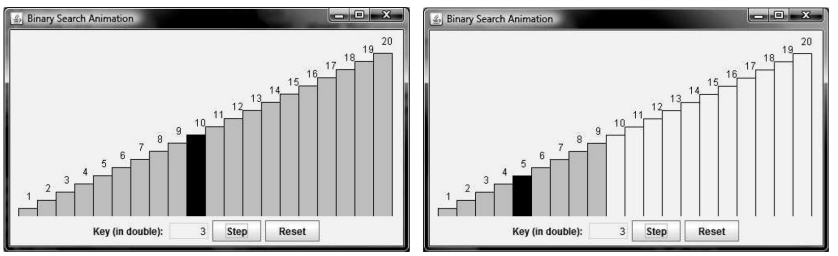
```
            executionTime[i] = System.currentTimeMillis() - startTime;
        }

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

    public static void findPrimeNumbers1(int N) {
        // final int N = 100000; // Odredi sve proste brojeve <= N
        final int NUMBER_PER_LINE = 10; // Prikazi 10 brojeva po liniji
        int count = 0; // Brojac za proste brojeve
        int number = 2; // Broj sa kojim delimo ispitanika

        //  System.out.println("Prosti brojevi su \n");

        // Jedan za drugim odredi proste brojeve
        while (number < N) {
            // Pretpostavimo da je broj prost
            boolean isPrime = true; // Da li je trenutni broj prost?

            // Provera da li je broj prost
            for (int divisor = 2; divisor <= (int) (Math.sqrt(number)); divisor++) {
                if (number % divisor == 0) { // Ako je true, broj nije prost
                    isPrime = false;
                    break; // Izadji iz for petlje
                }
            }

            // Odstampaj prost broj i uvecaj count za jedan
            //      if (isPrime) {
            //      count++; // Uvecaj count za 1
            //
            //      if (count % NUMBER_PER_LINE == 0) {
            //          // Odstampaj broj i predji u sledecu liniju
            //          System.out.println(number);
            //      }
            //      else {
            //          System.out.print(number + " ");
            //      }
            //      }

            // Proveri da li je sledeci broj prost
            number++;
        }
    }

    public static void findPrimeNumbers2(int N) {
        // Lista koja ce da cuva proste brojeve
        java.util.List<Integer> list = new java.util.ArrayList<Integer>();
        // final int N = 10000000; // Pronadji proste brojeve koji su  <= N
        final int NUMBER_PER_LINE = 10; // Prikazi po 10 brojeva u jednoj liniji
        int count = 0; // Brojac za proste brojeve
```

```
        int number = 2; // Broj sa kojim delimo ispitanika

        //  System.out.println("Prosti brojevi su \n");

        // Idi redom i pokusaj da nadjes prost broj
        while (number < N) {
            // Pretpostavimo da je broj prost
            boolean isPrime = true; // Da li je trenutni broj prost?

            // Provera da li je broj prost
            for (int k = 0; k < list.size() &&
                    list.get(k) <= (int) (Math.sqrt(number)); k++) {
                if (number % list.get(k) == 0) { // Ako je true, broj nije prost
                    isPrime = false; // Setuj isPrime na false
                    break; // Izadji iz for petlje
                }
            }

            // Proveri da li je sledeci broj prost
            number++;
        }
    }
}
```

ANIMACIJA BINARNOG PRETRAŽIVANJA (25 MIN)

Cilj programa je kreiranje animacije binarnog pretraživanja

Zadatak 2. Napisati Java FX program 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 5.2. 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 dijalog 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). (25 min)



Slika 5.2 Progam za prikaz animacije binarnog pretraživanja [1]

```
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);

    // Kreiraj scene i postavi je na stage
    Scene scene = new Scene(borderPane, 400, 250);
    primaryStage.setTitle("Exercise22_18"); // Postavi Naslov stage-a
    primaryStage.setScene(scene); // Postavi scenu na stage
    primaryStage.show(); // Prikazi 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()); // pronadjeno
    }
    else if (status == -1)
      lblStatus.setText("Kljuc pretrage nije u nizu");
  });

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

/**
 * Metod main je samo potreban za IDE sa limiranom JavaFX podrskom.
 * Nije neophodan za pokretanje iz komandne linije.
 */
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; // Pretrazi drugu polovini
        return 0;
      }
      else { // key < list[mid]
        high = mid - 1; // Pretrazi prvu polovini
        return 0;
      }
    }
  }

  class HistogramPane extends Pane {
    private int[] numbers;
    private int coloredBarIndex = -1;
    private int low = 0;
    private int high = -1; // Nemoj inicijaljno da prikazujes low i high

    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] + ""));
      }
    }
```

```
  }
}
```

Poglavlje 6

Zadaci za samostalan rad

ZADACI ZA SAMOSTALNO VEŽBANJE

Na osnovu materijala sa predavanja i vežbi uraditi samostalno sledeće primere

Zadatak 1. (10 min) (Maksimalni uzastopni rastuće uređeni podniz) Napišite program koji traži od korisnika da unese string i prikazuje maksimalan uzastopni rastuće uređen podniz. Analizirajte vremensku složenost vašeg programa. Evo primera rada:

Enter a string: Welcome (Enter)

Wel

Zadatak 2. (15 min) (Maksimalni rastuće uređeni podniz) Napišite program koji traži od korisnika da unese string i prikazuje maksimalan rastuće uređen podniz znakova. Analizirajte vremensku složenost vašeg programa.

Evo primera rada:

Unesite string: Welcome(Enter)

Welo

Zadatak 3. (15 min) (Pretraživanje podstringa - pattern matching) Napišite program koji traži od korisnika da unese dva niza i testira da li je drugi niz podniz prvog stringa. Pretpostavimo da su susedni karakteri u nizu različiti. (Nemojte koristiti metod indekOf u klasi String.) Analizirajte vremensku složenost vašeg algoritma. Vaš algoritam treba da bude reda O(n). Evo primera izvršavanja programa:

Enter a string s1: Welcome to Java (Enter)

Enter a string s2: come (Enter)

matched at index 3

Zadatak 4. (10 min) (Pronađi najmanji broj) Napišite metodu koja koristi pristup zavadi pa vladaj da pronađe najmanji broj na listi..

Zadatak 5. (10 min) (Svi prosti brojevi do 10.000.000.000) Napišite program koji pronalazi sve proste brojeve do 10.000.000.000. Postoji otprilike 455.052.511 takvih prostih brojeva. Vaš program treba da skladišti proste brojeve u binarnoj datoteci pod nazivom PrimeNumbers.dat. Kada se pronađe novi prost broj, broj se dodaje datoteci.

Zadatak 6. (10 min) Napišite program koji pronalazi broj prostih brojeva koji su manji ili jednaki 10, 100, 1.000, 10.000, 100.000, 1.000.000, 10.000.000, 100.000.000, Vaš program treba da pročita podatke sa PrimeNumbers.dat. Imajte na umu da datoteka sa podacima može nastaviti da raste kako se više prostih brojeva čuva u datoteci.

Zadatak 7. (30 min) (Animacija linearnog pretraživanja) Napišite Java applet koji animira algoritam linearne pretrage. Napravite niz koji se sastoji od 20 različitih brojeva od 1 do 20 u slučajnom redosledu. Elementi niza su prikazani u histogramu. Potrebno je da unesete ključ za pretragu u polje za tekst. Pritiskom na dugme Step izaziva program da izvrši jedno poređenje u algoritmu i ponovo oslika histogram trakom koja označava poziciju pretrage. Ovo dugme takođe zamrzava tekstualno polje kako bi sprečilo promenu njegove vrednosti. Kada se algoritam završi, prikažite okvir za dijalog da obavestite korisnika. Klikom na dugme Reset kreiraće se novi nasumični niz za novi početak. Ovo dugme takođe omogućava uređivanje tekstualnog polja.

Zadatak 8. Napisati aplikaciju koja animira problem 8 kraljica.

Poglavlje 7

Domaći zadatak

DOMAĆI ZADATAK - PRAVILA

Posebno obratiti pažnju na sledeća pravila u vezi izrade domaćih zadataka

Svaki student dobija od asistenta sopstvenu kombinaciju domaćeg zadatka.

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

Odgovarajući NetBeans (Eclipse ili Visual Studio) projekat koji predstavlja rešenje domaćeg zadatka smestiti u folder CS203-DZ03-Ime-Prezime-BrojIndeksa. Zipovani folder CS203-DZ03-Ime-Prezime-BrojIndeksa poslati predmetnom asistentu (lazar.mrkela@metropolitan.ac.rs) u mejlu sa naslovom (subject) CS203-DZ03, inače se neće računati.

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

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

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

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

Vreme za izradu: 2h.

Zaključak

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće:

U lekciji je poseban akcenat stavljen na vrste pretraživanja nizova, kao i na implementaciju i analizu vremenske složenosti binarnog pretraživanja. Pored toga, kroz primere je prikazano dinamičko programiranje, algoritam Sito Eratostena, objašnjen je problem određivanja maksimalne sume podsekvence niza uz primenu poboljšanih verzija početnog algoritma. Na kraju je prikazano rešenje problema 8 kraljica primenom obrnute pretrage.

REFERENCE

Korišćena literatura

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

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

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

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