

# Poglavje 1

## Pregled metod reševanja

### 1.1 Deli in vladaj

Pri metodi deli in vladaj problem delimo na podprobleme, dokler jih ne znamo rešiti direktno, nato pa te delne rešitve sestavimo v končno rešitev. Navadno je take algoritme najlažje implementirati rekurzivno.

Nekaj problemov, ki jih rešujemo s to metodo:

- Urejanje (QuickSort, MergeSort)
- Iskanje v urejenem seznamu (binarno iskanje)
- Množenje števil (Karatsubov algoritem)
- Množenje matrik (Strassenov algoritem)

### 1.2 Dinamično programiranje

Dinamično programiranje je v osnovi podobno metodi deli in vladaj, saj podobno problem delimo na podprobleme in nato sestavljamo končno rešitev, s to razliko, da shranjujemo vmesne rezultate. Če so podproblemi odvisni med seboj, ali na delih celo enaki, je dinamično programiranje, v primejavi z metodo deli in vladaj hitrejše, a prostorsko zahtevnejše.

Nekaj problemov, ki jih rešujemo s to metodo:

- Izračun členov Fibonaccijevega zaporedja
- Reševanje igre Hanojski stolpi
- Neomejeni problem nahrbtnika

### 1.3 Požrešna metoda

Pri požrešni metodi se, kadar imamo na voljo večih vej, odločimo za lokalno najboljšo in pri tem ne hranimo prejšnjih odločitev, ter se ne vračamo nazaj.

Nekaj problemov, ki jih rešujemo s to metodo:

Pri težkih problemih s požrešnimi algoritmi v splošnem ne dobimo optimalne rešitve, se ji pa lahko približamo, če uporabljamo pametne heuristike:

- Problem trgovskega potnika
- Neomejeni problem nahrbtnika

Obstaja pa precej problemov, kjer lokalno najboljša izbira vodi tudi do končne najboljše rešitve:

- Preprosti problem nahrbtnika
- Iskanje najkrajših poti v grafu (Dijkstrov algoritem)
- Iskanje minimalnega vpetega drevesa (Primov in Kruskalov algoritem)
- Izgradnja optimalnega drevesa pri Huffmanovem kodiranju.

## 1.4 Sestopanje in razveji & omeji

Sestopanje je preiskovanje celotnega prostora rešitev in najde vse možne rešitve. S pomočjo heuristik pa lahko včasih odrežemo določene veje, če ugotovimo da tam rešitve zagotovo ni, ali pa če iščemo točno določeno optimalno rešitev. V najslabšem primeru lahko ti metodi vseeno preiščeta celoten prostor rešitev.

Nekaj problemov, ki jih rešujemo s to metodo:

- Problem trgovskega potnika
- Problem popolnega pokritja (npr. izdelava križanke, problem osmih dam, ...)
- Problem skakačevega obhoda

## 1.5 Linearno programiranje

Linearno programiranje je nastalo že pred računalniškim programiranjem in je metoda za iskanje največje vrednosti kriterijske funkcije, glede na omejitve, podane z linearnimi neenačbami.

### 1.5.1 Simpleksni algoritem

Algoritem rešuje linearne programe tako, da se sprehaja po ogliščih konveksne poliederske množice (KPM), ki jo tvori presek danih neenačb (v 2D je KPM konveksen poligon, v 3D polieder, ...). Pri sprehajanju se vedno premikamo le proti boljšim rešitvam, če to v nekem oglišču ni možno, pa smo dosegli maksimum in se algoritem ustavi.

Predpostavlja se, da začenjamo iz koordinatnega izhodišča ( $\forall x_i = 0$ ), kadar pa izhodišče ne pripada KPM, koordinatni sistem ustrezno preslikamo (premik, vrtenje, zrcaljenje, ...).

# Poglavje 2

# Teorija števil

## 2.1 Praštevila

Praštevila so naravna števila, ki so deljiva le z 1 in sama s seboj. Ostalim naravnim številom, razen 1, pravimo sestavljena števila, saj jih je moč sestaviti kot produkt praštevil.

Kot je dokazal že Evklid[?], obstaja neskončno praštevil. Uporabimo preprost dokaz z protislovjem. Predpostavimo, da imamo množico števil, ki vsebuje vsa praštevila  $P = \{2, 3, 5, \dots, p\}$ . Če sedaj vsa praštevila zmožimo in na koncu prištejemo 1, dobimo število, ki ga ne deli nobeno izmed teh praštevil. Ker obstajajo le praštevila in števila sestavljena iz prafaktorjev, lahko sklepamo, da je to število novo praštevilo, ki ga ni v naši začetni množici, to pa je v protislovju s predpostavko. V resnici ni nujno, da je dobljeno število praštevilo – druga možnost je, da obstaja še kako praštevilo, ki je manjše od zmnožka vseh praštevil, ki smo jih zajeli v množici  $P$ .

$(2) + 1$	$=$	3	(je praštevilo)
$(2 * 3) + 1$	$=$	7	(je praštevilo)
$(2 * 3 * 5) + 1$	$=$	31	(je praštevilo)
$(2 * 3 * 5 * 7) + 1$	$=$	211	(je praštevilo)
$(2 * 3 * 5 * 7 * 11) + 1$	$=$	2311	(je praštevilo)
$(2 * 3 * 5 * 7 * 11 * 13) + 1$	$=$	30031	$= 59 * 509$ (ni praštevilo)
...		...	...

### 2.1.1 Preprosti algoritem za ugotavljanje praštevilstosti

Algoritem preveri, ali je parameter  $a$  deljiv s katerim izmed naravnih števil v intervalu  $[2, \sqrt{a}]$  in če takega števila ni, zaključimo, da je  $a$  praštevilo. Razlog, zaradi katerega lahko praštevilstost preverimo s tem intervalom (kljub temu, da bi tudi  $\frac{a}{2}$  lahko delil  $a$ ), leži v tem, da če je parameter sestavljen, je največji možen razcep na prafaktorje  $\sqrt{a} * \sqrt{a}$ .

Preprosti algoritem za preverjanje praštevilstosti

```

1 public static boolean isPrime(int a) {
2     if(a<=1) return false;
3
4     for(int i=2; i<=Math.sqrt(a); i++)
5         if(a%i==0) return false;
6
7     return true;
8 }
9
10 public static boolean isPrime2(int a) {
11     return BigInteger.valueOf(a).isProbablePrime(5);
12 }
```

## 2.2 Največji skupni delitelj in najmanjši skupni večkratnik

Največji skupni delitelj (angl. greatest common divisor – GCD), je največje naravno število, ki deli vsa dana števila.

### 2.2.1 Evklidov algoritem

Evklidov algoritem za izračun GCD je verjetno eden najstarejših algoritmov, ki so še vedno v uporabi. Prvi ohranjen zapis je v Evklidovih Elementih[?], že prej pa naj bi bil znan Pitagorejcem.

Algoritem temelji na ugotovitvi, da se GCD ne spremeni, če od večjega števila odštejemo manjšega. Če to odštevanje ponavljamo, dokler ena izmed spremenljivk ne doseže 0, je vrednost druge spremenljivke ravno GCD. Kasneje so ugotovili, da isto velja tudi za ostanek po deljenju, le da je potrebno precej manj korakov za izračun, zato raje uporabljamo ta način.

Evklidov algoritem za iskanje največjega skupnega delitelja

```

1 public static int GCD(int a, int b) {
2     if(a==0) return b; else
3     if(b==0) return a; else
4     if(a>b) return GCD(b, a%b); else
5     return GCD(a, b%a);
6 }
7
8 public static int GCD2(int a, int b) {
9     return BigInteger.valueOf(a).gcd(BigInteger.valueOf(b)).intValue();
10 }
```

### 2.2.2 Najmanjši skupni večkratnik

Najmanjši skupni večkratnik (angl. least common multiple (LCM)) je najmanjše naravno število, ki je večkratnik danih naravnih števil.

Lahko ga izračunamo iz največjega skupnega delitelja po preprosti formuli:

$$\text{lcm}(a, b) = \frac{a * b}{\text{gcd}(a, b)}$$

## 2.3 Fibonaccijevo zaporedje

Fibonacci je eden izmed najbolj znanih srednjeveških matematikov, ki je, med drugim, pisal tudi o zaporedju, ki je kasneje dobilo ime po njem[?]. Zaporedje je predstavil na primeru idealizirane rasti populacije zajcev:

Recimo, da vsak par zajcev živi dve sezoni in vsako sezono ustvari en par potomcev. Imamo neko pokrajino, kjer zajcev še ni in začnemo z enim mladim parom in nato vsako sezono preštejemo nove pare. Čez eno sezono imamo en nov par, v drugi sezoni imajo potomce prva in druga generacija, tako da imamo dva nova para, hkrati pa prva generacija odmr. Naslednjo sezono imamo tri nove pare, umre pa druga generacija. Preostali pari ustvarijo pet novih parov, umre tretja generacija. . .

Vidimo, da je število potomcev v resnici enako velikosti zadnjih dveh generacij.

Prvih nekaj členov Fibonaccijevega zaporedja je tako: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . . , kar lahko zapišemo z rekurzivno enačbo:

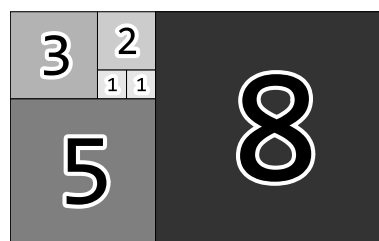
$$F(n) = \begin{cases} 0 & ; \quad n = 0, \\ 1 & ; \quad n = 1, \\ F(n-1) + F(n-2) & ; \quad n \geq 2. \end{cases}$$

Kvocien členov Fibonaccijevega zaporedja konvergira proti zlatemu rezu. To je moč prikazati tudi grafično, če člene zaporedja narišemo kot kvadrate ustreznih velikosti - dobimo namreč pravokotnik, katerega razmerje stranic se z dodajanjem novih členov približuje zlatemu rezu:

Člena	Kvocien
1, 2	2
2, 3	1.5
3, 5	1.666
5, 8	1.6
...	...
987, 610	1.618033...

Zlati rez:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618034...$$



Možna posplošitev zaporedja je t.i. Fibonaccijevo zaporedje višjega reda. Če spet vzamemo Fibonaccijevo zgodbo z zajci, zaporedje višjega reda pomeni le, da zajci živijo  $k$  let, namesto dveh. Pri zaporedju  $k$ -tega reda, torej seštejemo prejšnjih  $k$  členov zaporedja.

Rekurzivno enačbo za Fibonaccijevo zaporedje  $k$ -tega reda zapišemo zapišemo kot:

$$F(n) = \begin{cases} 0 & ; \quad n < k; \\ 1 & ; \quad n = k; \\ F(n-1) + F(n-2) + \dots + F(n-k) & ; \quad n > k. \end{cases}$$

### 2.3.1 Algoritem za izračun prvih $n$ členov Fibonaccijevega zaporedja

Pri tem algoritmu najprej nastavimo prvih  $k-1$  členov na nič in  $k$ -tega na ena, nato pa nove člene izračunamo kot vsoto zadnjih  $k$  členov zaporedja.

Algoritem za izračun prvih  $n$  števil Fibonaccijevega zaporedja reda  $k$

```

1 public static int[] Fib(int n, int k) {
2     // tabela vrednosti zaporedja (na začetku 0,0,...,0,1,0,...)
3     int[] out = new int[n];
4     out[k-1]=1;
5
6     // generiramo zaporedje naprej - nov element je vsota zadnjih k elementov
7     for(int i=k; i<n; i++)
8         for(int j=0; j<k; j++)
9             out[i] += out[i-j-1];
10
11     return out;
12 }
```

### 2.3.2 Algoritem za ugotavljanje pripadnosti števila Fibonaccijevemu zaporedju

Algoritem računa člene Fibonaccijevega zaporedja  $k$ -tega reda in se ustavi, ko naleti na člen, ki je večji ali enak podanemu parametru. Pri računanju potrebujemo le zadnjih  $k$  členov, ki pa bi jih bilo dobro hraniti tako, da nam ne bi bilo treba kakorkoli premetavati elementov tabele. Tu lahko uporabimo krožno tabelo, ki jo lahko simuliramo z navadno tabelo, če elemente naslavljamo po modulu velikosti tabele. Za izračun novega elementa bi lahko vedno znova računali vsoto vseh elementov, a to ni potrebno, saj je pri Fibonaccijevem zaporedju naslednji člen na položaju  $i$  v tabeli mogoče preprosto izračunati:

$$\begin{aligned}\text{naslednjiClen} &= 2 \cdot \text{trenutniClen} - \text{tabela}[i] \\ \text{tabela}[i] &= \text{trenutniClen}\end{aligned}$$

Na primeru Fibonaccijevega zaporedja tretjega reda računamo tako:

Elementi tabele	Stara vsota	Nova vsota
0 0 1	—	1
<u>1</u> 0 1	1	2
1 <u>2</u> 1	2	4
1 2 <u>4</u>	4	7
<u>7</u> 2 4	7	13
7 <u>13</u> 4	13	24
7 13 <u>24</u>	24	44

Algoritem za preverjanje, ali je število del Fibonaccijevega zaporedja reda  $k$

```
1 public static boolean isFib(int a, int k) {
2     if (a==0) return true;
3
4     // tabela zadnjih k vrednosti zaporedja (na začetku 0,0,...,0,1)
5     int[] store = new int[k];
6     store[k-1]=1;
7     // vsota členov
8     int sum=1;
9
10    // kaže na mesto v tabeli, ki ga bomo prepisali
11    int pos=0;
12
13    while(a>sum) {
14        int oldSum = sum;
15        sum = 2*sum-store[pos%k];
16        store[pos%k] = oldSum;
17
18        pos++;
19    }
20
21    return (sum==a); // ali je a člen zaporedja
22 }
```

## Poglavje 3

# Iskalni algoritmi

### 3.1 Binarno iskanje

Binarno iskanje je učinkovit algoritem za iskanje v urejeni tabeli.

Algoritem temelji na strategiji deli ni vladaj in sicer deluje tako, da na vsakem koraku razpolovi velikost intervala tabele, v katerem išče. To stori tako, da primerja element na sredini intervala, s ključem, ki

se išče. Če je sredinski element manjši od ključa, naslednik sredinskega elementa v tabeli postane nova spodnja meja, če je sredinski element večji od ključa, pa je naslednik sredinskega elementa v tabeli nova zgornja meja. Iskanje se zaključi, ko naletimo na element, ki je enak ključu, če takega elementa v tabeli ni, pa se iskanje zaključi, ko se zgornja in spodnja meja intervala preiskovanja prekrijeta.

Algoritem binarnega iskanja oz. bisekcije v urejeni tabeli

```

1  public static int binarySearch(int a[], int key) {
2      int left = 0, mid, right = a.length-1;
3
4      while(left<=right) {
5          mid = (left+right)>>>1; // sredina intervala
6          if(a[mid]<key) left = mid+1; else // premaknemo spodnjo mejo
7          if(a[mid]>key) right = mid-1; else // premaknemo zgornjo mejo
8          return mid; // našli smo iskani element
9      }
10     return -1; // iskanega elementa ni v tabeli
11 }
12
13 public static int binarySearch2(int a[], int key) {
14     return Arrays.binarySearch(a, key);
15 }

```

## Poglavje 4

# Grafi

Graf formalno podamo kot dvojček vozlišč  $V$  in povezav  $E$  med vozlišči:  $G = \langle V, E \rangle$ .

Navadno porabimo enega od naslednjih zapisov grafov:

- **Seznam sosednosti** – za vsako vozlišče hranimo seznam povezanih vozlišč (omogoča hrambo dodatnih podatkov v vozliščih)
- **Seznam pojavnosti** – za vsako vozlišče hranimo seznam njegovih povezav in za vsako povezavo njen seznam vozlišč (omogoča hrambo dodatnih podatkov v vozliščih in na povezavah)
- **Matrika sosednosti** – kvadratna matrika vozlišč, kjer  $M_{i,j} \neq 0$  pomeni povezavo med vozliščema  $v_i$  in  $v_j$ . (omogoča hrambo ene vrednosti na povezavah)
- **Matrika pojavnosti** – matrika vozlišč, kjer  $M_{i,j} \neq 0$  pomeni povezavo med vozliščem  $v_i$  in povezavo  $e_j$ . (omogoča hrambo ene vrednosti na povezavah)

### 4.1 Eulerjev in Hamiltonov graf

#### 4.1.1 Eulerjev graf

Graf je Eulerjev, kadar vsebuje Eulerjev cikel – tak sprehod, ki vsako povezavo uporabi natanko enkrat in se na koncu vrne v izhodišče. Multigraf je nadgradnja navadnih grafov s tem, da dovolimo več vzporednih povezav med dvema vozliščema in dovolimo tudi povezave vozlišč samih vase. Tudi multigraf je lahko Eulerjev. Preden poženemo algoritem lahko preverimo če graf izpolnjuje potrebne pogoje, npr. ali je povezan in ali so vsa vozlišča sode stopnje.

Pseudokoda Fleuryjevega algoritma za iskanje Eulerjevega cikla

```

0  izberi začetno vozlišče
1  prečkaj poljubno povezavo in jo odstrani (tu most prečkamo le če ni boljše izbire)
2  če je bil odstranjen most, odstrani vse točke ki so ostale izolirane
3  če je še kaka povezava, pojdi na 1
4  sicer zaključi algoritem.

```

### 4.1.2 Hamiltonov graf

Graf je Hamiltonov, kadar vsebuje Hamiltonov cikel – tak sprehod, ki vsako vozlišče obišče natanko enkrat in se na koncu vrne v izhodišče.

## 4.2 Iskanje maksimalnega pretoka skozi graf

Predstavljamo si, da imamo nek vir tekočine, ki teče v razvejan sistem cevi, v katerem ima vsaka cev določen maksimalen pretok tekočine ( $c_{i,j}$ ), ki ga lahko prenese med dvema točkama. Vsaka cev se steka v neko točko in iz vsake točke lahko naprej vodi več cevi. Tu seveda velja pravilo, da iz vsakega vozlišča odteka enako tekočine kot vanjo pride (tekočina se ne more nikjer zadrževati). Sistem ima poleg izvora tekočine tudi t.i. ponor, kamor se tekočina na koncu izteka. Zanima nas, največ koliko tekočine lahko spravimo skozi tak sistem cevi od izvora do ponora.

Sistem cevi predstavimo kot graf, kjer je izvor označen z prvim indeksom in ponor z zadnjim, nato pa po nekem algoritmu spreminjamo pretoke skozi cevi ( $x_{i,j} \leq c_{i,j}$ ), dokler ne najdemo maksimalnega.

### 4.2.1 Ford-Fulkersonov algoritem

Ideja algoritma je, da iščemo poti od izvora do ponora, ki še lahko sprejejo več tekočine in povečamo pretok do njih. Če take poti ni moč najti, smo našli optimalen pretok in se algoritem zaključi. Pravimo, da je celotna pot je zasičena (ima največji možen pretok), če je vsaj ena povezava na njej zasičena, nezasičena pa kadar so vse povezave na njej nezasičene. Povezava je zasičena, ko po njej ne moremo povečati pretoka, torej  $x_{i,j} = c_{i,j}$ .

Kadar potujemo po pozitivni (pravilno usmerjeni) povezavi, hočemo zagotoviti maksimalen pretok ( $x_{i,j} = c_{i,j}$ ), lahko pa potujemo tudi po negativnih povezavah (obrnjenih proti toku), edina razlika je, da v tem primeru želimo minimalen pretok ( $x_{i,j} = 0$ ).

Psevdokoda Ford-Fulkersonovega algoritma za iskanje maksimalnega pretoka skozi graf

```

0 vse pretoke nastavi na 0
1 označi izvirno vozlišče
2 iz označenih vzemi poljubno vozlišče in ga zaznamuj kot obiskanega
3 označi vse neobiskane sosedo do katerih obstaja nezasičena povezava
4 če je povečanje (ali zmanjšanje) toka po povezavi po kateri smo prišli v trenutno vozlišče ←
  manjše od shranjenega, ga shrani
5 če obstajajo označena vozlišča in ponor še ni označen, pojdi na 2
6 če je ponor označen, pojdi po poti od ponora nazaj in popravi pretoke s shranjenim in pojdi ←
  na 1
7 če ni več označenih vozlišč, zaključi algoritem.
```

## 4.3 Topološko urejanje

Če je usmerjen graf aciklični ga je moč urediti tako, da vozlišča z nižjo oznako vedno kaže le na vozlišča z višjimi oznakami. Tej ureditvi pravimo topološka ureditev.

Psevdokoda algoritma za topološko urejanje grafa

```

0 naredi delovno kopijo grafa
1 poišči vozlišče brez vhodnih povezav
2 če takega vozlišča ni, zaključi algoritem z napako. (odkrili smo cikel)
3 sicer vozlišče označi z naslednjo oznako in ga odstrani iz grafa
4 če obstaja še kako vozlišče, pojdi na 1
5 sicer prenesi oznake na prvotni graf in zaključi algoritem.
```

## 4.4 Iskanje najcenejših poti

### 4.4.1 Dijkstraov algoritem

Algoritem najde najcenejšo pot iz nekega začetnega vozlišča, do vseh ostalih vozlišč v grafu.

### Pseudokoda Dijkstovega algoritma za iskanje najcenejših poti

```
0 izberi začetno vozlišče, označi ga kot obiskanega in nastavi njegovo trenutno razdaljo na 0
1 ostala vozlišča označi kot neobiskana, nastavi trenutne razdalje na neskončno in jih dodaj v ←
  vrsto neobiskanih
2 izračunaj dolžine poti od izbranega vozlišča do neobiskanih sosedov
3 če je dobljena dolžina do sosedu kje manjša od že znane, jo shrani
4 izbrano vozlišče označi kot obiskano in ga odstrani iz vrste neobiskanih (razdalja do tu je ←
  že minimalna)
5 če vrsta neobiskanih ni prazna, izberi vozlišče z najmanjšo trenutno razdaljo in pojdi na 3
6 sicer zaključi algoritem.
```

## Poglavje 5

# Kombinatorika in verjetnost

### 5.1 Permutacije

#### Algoritem za izračun permutacij danega niza

```
1 // Izpiši permutacije (v pravem zaporedju)
2 public static void perm1(String s) {
3     perm1("", s);
4 }
5 private static void perm1(String prefix, String s) {
6     if(s.length()==0) {
7         System.out.println(prefix);
8     } else for(int i=0; i<s.length(); i++) {
9         perm1(prefix + s.charAt(i), s.substring(0, i) + s.substring(i+1, s.length()));
10    }
11 }
12
13 // Izpiši permutacije (niso v pravem zaporedju)
14 public static void perm2(String s) {
15     char[] a = new char[s.length()];
16     for(int i=0; i<s.length(); i++)
17         a[i] = s.charAt(i);
18
19     perm2(a, s.length());
20 }
21 private static void perm2(char[] a, int len) {
22     if (len==1) {
23         System.out.println(a);
24     } else for(int i=0; i<len; i++) {
25         swap(a, i, len-1);
26         perm2(a, len-1);
27         swap(a, i, len-1);
28     }
29 }
```

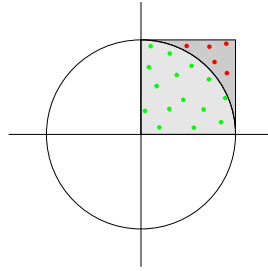
### 5.2 Metoda Monte Carlo

Monte Carlo je verjetnostna metoda, ki deluje tako, da vzamemo neko množico v prostoru, ki je večja od dane množice katere velikost želimo oceniti, znamo pa ugotoviti, ali je neka točka znotraj te množice. Nato naključno izbiramo točke v naši množici in preverjamo ali so znotraj dane množice ali ne. Končna ocena metode je  $\frac{\text{zadeti}}{\text{poskusi}}$ .

#### 5.2.1 Izračun približka števila $\pi$

V koordinatnem izhodišču narišemo krog s polmerom 1 in kvadrat s stranico 1 v prvem kvadrantu. Če kot zadetek štejemo točke, ki so za največ 1 oddaljene od izhodišča dobimo približek za  $\frac{\pi}{4}$ .





Algoritem za izračun približka števila  $\pi$  po metodi Monte Carlo

```

1 public static double monteCarloPi(int num) {
2     Random rand=new Random();
3     int hits=0;
4     for(int i=0; i<num; i++) {
5         double x=rand.nextDouble(), y=rand.nextDouble();
6
7         if(Math.sqrt(x*x + y*y)<=1) hits++;
8     }
9
10    return ((double)hits/num)*4;
11 }

```

Taka metoda za iskanje dobrega približka števila  $\pi$  ni najbolj primerna, je pa preprost primer uporabe.

## Poglavje 6

# Računska geometrija

### 6.1 Najbližji par v množici točk

Želimo poiskati točki, ki sta si najbližji v neki množici točk. Naivni algoritem bi bil pregled vseh parov, kar bi zahtevalo kvadratno časovno zahtevnost, z boljšimi metodami pa dosežemo zahtevnost  $O(n \log(n))$ . Poglejmo preprost algoritem s pometanjem, ki ima v najboljšem primeru linearno zahtevnost, v najslabšem pa kvadratno.

Algoritem za iskanje najbližjega para točk v množici točk

```

1 static int[][] tocke;
2
3 private static int[] closestPair() {
4     //uredi točke po Y
5     sort(tocka);
6
7     int minTocka1 = 0, minTocka2 = 1;
8     double minRazd = razdalja(minTocka1, minTocka2);
9     //ugotovi najkrajšo razdaljo med sosednjimi točkami po dimenziji Y
10    //ta korak ni nujen, ampak lahko pohitri računanje, ker prej dobimo
11    //boljšo trenutno vrednost za najkrajšo razdaljo
12    for(int i = 1; i < tocke.length-1; i++) {
13        int t1 = i, t2 = i+1;
14        if(((tocke[t2][1]-tocke[t1][1]) >= minRazd)
15            ||(Math.abs(tocke[t2][0]-tocke[t1][0]) >= minRazd)) continue;
16
17        double tmpRazd = razdalja(t1, t2);
18        if(tmpRazd < minRazd) {
19            minRazd = tmpRazd;
20            minTocka1 = t1;
21            minTocka2 = t2;
22        }
23    }
24 }

```

```

24
25 //preišče okno točk od t1+1...t2 v katerem so lahko točke,
26 //ki so bližje od trenutne vrednosti za najkrajšo razdaljo
27 int t1 = 0, t2 = 2;
28 do {
29     do {
30         double tmpRazd = razdalja(t1, t2);
31         if(tmpRazd < minRazd) {
32             minRazd = tmpRazd;
33             minTocka1 = t1;
34             minTocka2 = t2;
35         }
36         do t2++; while((t2 < tocke.length)
37             &&(Math.abs(tocke[t2][0]-tocke[t1][0]) >= minRazd)
38             &&((tocke[t2][1]-tocke[t1][1]) < minRazd));
39     } while((t2 < tocke.length)&&((tocke[t2][1]-tocke[t1][1]) < minRazd));
40     do {
41         t1++;
42         t2 = t1+2;
43     } while((t2 < tocke.length)&&((tocke[t2][1]-tocke[t1][1]) >= minRazd));
44 } while(t2 < tocke.length);
45
46 return new int []{minTocka1, minTocka2};
47 }
48
49 private static double razdalja(int t1, int t2) {
50     return Math.sqrt((long)(tocke[t1][0]-tocke[t2][0])*(tocke[t1][0]-tocke[t2][0])
51         + (long)(tocke[t1][1]-tocke[t2][1])*(tocke[t1][1]-tocke[t2][1]));
52 }

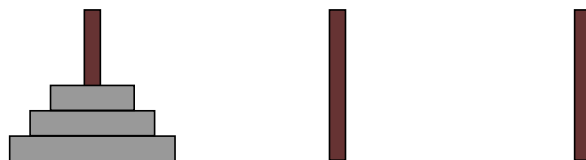
```

## Poglavje 7

# Random stuff

### 7.1 Hanojski stolpi

Hanojski stolpi je igra, ki jo sestavljajo tri palice in nekaj diskov različnih velikosti. Na začetku so diski urejeni od najmanjšega do največjega po velikosti na prvi palici, cilj igre pa je premakniti vse diske s prve palice na drugo, pri čemer ne moremo položiti večjega diska na manjšega in s palice lahko dvignemo le zgornji disk.



#### 7.1.1 Najmanjše število premikov glede na število diskov

Če hočemo iz začetnega položaja premakniti prvi(zgornji) disk, to lahko storimo z enim premikom, če pa želimo premakniti disk pod njim, moramo najprej premakniti prvega, šele nato drugega. Po teh dveh premikih je za nadaljevanje igre potrebno eno palico sprostiti, kar dosežemo s premikom prvega diska na drugega - skupaj trije premiki. Če bi iz začetnega položaja želeli premakniti tretji disk, bi najprej s tremi premiki premaknili prva dva diska na tretjo palico, tretji disk na drugo palico, potem pa spet s tremi premiki prvi in drugi disk na drugo palico - skupaj sedem premikov. Za premik četrtega diska pa bi najprej premaknili prve tri, nato četrtega in prve tri na četrtega - petnajst premikov. Splošna formula za število premikov je:  $\text{moves} = 2^n - 1$

### 7.1.2 Rekurzivni algoritem za reševanje igre Hanojski stolpi

Algoritem pokličemo s številom diskov, nato pa algoritem v vsakem klicu opravi tri stvari:

- rekurzivno premakne vse manjše diske na eno od palic
- premakne trenutni disk
- rekurzivno premakne vse manjše diske na trenutni disk

Rekurzivni algoritem za reševanje igre Hanojski stolpi

```
1 public static void towersOfHanoi(int n, int source, int dest, int by) {
2     if(n==1) {
3         System.out.println(source+"->"+dest);
4     } else {
5         towersOfHanoi(n-1, source, by, dest);
6         towersOfHanoi(1, source, dest, by);
7         towersOfHanoi(n-1, by, dest, source);
8     }
9 }
```

## 7.2 CRC kodiranje

Pri CRC(Cyclic Redundancy Check) varnostnem kodiranju zaporedje bitov, delimo z vnaprej znanim zaporedjem bitov t.i. polinomom. Rezultat ki ga iščemo je ostanek po deljenju.

Algoritem za izračun CRC varnostnih bitov

```
1 public static boolean[] CRC(boolean[] bits, boolean[] poly) {
2     // ustvarimo dovolj veliko tabelo za izračun in na začetek prilepimo vhodne bite
3     boolean[] tmpbits = new boolean[bits.length + poly.length];
4     System.arraycopy(bits, 0, tmpbits, 0, bits.length);
5
6     for(int pos=0; pos<bits.length; pos++) if(tmpbits[pos])
7         for(int i=0; i<poly.length; i++) if(poly[i])
8             tmpbits[pos+i] = tmpbits[pos+i]^poly[i];
9
10    boolean[] CRCbits = new boolean[poly.length-1];
11    System.arraycopy(tmpbits, bits.length+1, CRCbits, 0, poly.length-1);
12    return CRCbits;
13 }
```

## 7.3 Levenshteinova razdalja med nizi

Levenshteinova razdalja je mera podobnosti nizov, ki jo izračunamo kot najmanjše število določenih urejanj, po katerih pridemo od enega niza do drugega.

Vrste urejanj so:

- brisanje znaka:  $d_L("aaa", "aa") = 1$
- vstavljanje znaka:  $d_L("aaa", "aaaa") = 1$
- spremembo znaka:  $d_L("aaa", "aba") = 1$

Primer:

- dve spremembi in vstavljanje:  $d_L("Burek", "Baraka") = 3$

Algoritem za izračun Levensteinove razdalje med dvema nizoma

```
1 public static int levensteinDistance(String s1, String s2) {
2     int[][] d = new int[s1.length()+1][s2.length()+1];
3
4     for(int i=0; i<=s1.length(); i++) d[i][0] = i;
5     for(int j=0; j<=s2.length(); j++) d[0][j] = j;
6
7     for(int i=1; i<=s1.length(); i++) {
8         for(int j=1; j<=s2.length(); j++) {
9
10            int del = d[i-1][j]+1, // brisanje
11            ins = d[i][j-1]+1, // vstavljanje
12            sub = d[i-1][j-1]; // sprememba
```

```
13
14         if(s1.charAt(i-1) != s2.charAt(j-1)) sub++;
15
16         // shrani najmanjšo izmed razdalj
17         d[i][j] = Math.min(del, Math.min(ins, sub));
18     }
19 }
20
21 return d[s1.length()][s2.length()];
22 }
```