Diskrétní matematika

Petr Kovář
petr.kovar@vsb.cz

Vysoká škola báňská – Technická univerzita Ostrava

zimní semestr 2022/2023
DiM 470-2301/01, 470-2301/03*, 470-2301/05

O tomto souboru

Tento soubor je zamýšlen především jako pomůcka pro přednášejícího. Řadu důležitých informací v souboru nenajdete, protože přednášející je říká, ukazuje, případně maluje na tabuli. Přednášky jsou na webu k dispozici, aby studenti mohli snadno dohledat probíraná témata z přednášek, které zameškali.

Pro samostatné studium doporučuji skripta:

- M. Kubesa: Základy diskrétní matematiky, výukový text
- P. Kovář: Úvod do teorie grafů, výukový text

Pro přípravu ke zkoušce a písemkám doporučuji cvičebnici:

- P. Kovář: Cvičení z diskrétní matematiky, sbírka příkladů

Vše na http://homel.vsb.cz/~kov16/predmety_dm.php

Přehled přednášky

Algoritmizace diskrétních struktur

- programové interpretace struktur
- implementace množin
- generování výběrů
- generátory náhodných čísel
- kombinatorická exploze

7. Algoritmizace diskrétních struktur

Závěrem prvního tématického celku o diskrétních strukturách si povíme něco o jejich implementaci do programovacích jazyků.

Některé probrané struktury a postupy lze naprogramovat snadno, jiné obtížně. Existuje řada přístupů, které se liší v nároku na paměť a/nebo strojový čas.

Často obecný přístup bývá na úkor rychlosti, ale ne vždy musí být obecný přístup řádově pomalejší.

Tato kapitola je věnována vybraným implementacím.

7.1. Programová implementace struktur

- posloupnosti
- zobrazení
- relace
- permutace

(Konečnou) posloupnost $(a_0, a_1, \ldots, a_{n-1})$
implementujeme jako jednorozměrné pole `a[ ]`, kde `a[i] =` a_i.

Příklad

Máme dánu (konečnou) posloupnost $(7, 5, 5, 7, 5, 6, 6)$.
Uložíme ji do pole `p = [ 7 5 5 7 5 6 6 ]`.

Zobrazení

Zobrazení $f : A \to B$

Mějme konečné $A = \{a_0, a_1 \dots, a_{n-1}\}$ a $B = \{b_0, b_1, \dots, b_{m-1}\}$.
Pracujeme s indexy a implementujeme jako posloupnost – pole `f[ ]`, ve kterém `f[i]=j` vyjadřuje $f(a_i) = b_j$.

Šikovné, jsou-li A a B celočíselné obory a prvky jsou malá celá čísla. Pro jiné množiny musíme „překládat" prvky z A, B na jejich indexy (složité/náročné na strojový čas).

- pro prvky množiny A využijeme datové typy jako hašovací tabulky
- pro prvky množiny B využijeme pole strukturovaných proměnných či pointerů (ukazatelů)

Příklad

Máme dáno zobrazení $f : [0, 5] \to [0, 5]$, kde $f(0) = 4$, $f(1) = 5$, $f(2) = 3$, $f(3) = 3$, $f(4) = 2$, $f(5) = 2$.
Zobrazení uložíme do pole `f = [ 4 5 3 3 2 2 ]`.

Příklad

Máme dáno zobrazení $f : \{A, B, C, D, E\} \to \{x, y, z, w\}$,
kde $f(A) = w$, $f(B) = z$, $f(C) = w$, $f(D) = x$, $f(E) = w$.
Zobrazení uložíme do pole `f = [ 3 2 3 0 3 ]`.
Potřebujeme pomocná pole `X = [ A B C D E ]`, `Y = [ x y z w ]`.

Příklad

Máme dáno zobrazení $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, kde $f(x, y) = x^2 + 3y$.
Do pole uložit nejde! Nemá smysl ukládat $\mathbb{R}$.

Příklad

Máme dáno zobrazení $f : [-5 : 5] \times [-5 : 5] \to \mathbb{R}$, kde $f(x, y) = x^2 + 3y$.
Uložíme do dvourozměrného pole 11×11 (obecně přibližných!) hodnot.

Příklad

Máme dáno zobrazení $f : [-5 : 5] \times [-5 : 5] \to \mathbb{R}$, kde $f(x, y) = \sqrt{x + y}$.
Uložíme do dvourozměrného pole 11×11 (přibližných!) hodnot.

Relace – opakování

Definice

(Homogenní) binární relace R na množině A je libovolné podmnožina kartézského součinu $A \times A = A^2$, tj.

$$R \subseteq A^2.$$

Pokud $(x, y) \in R$, říkáme, že „prvek x je v relaci s prvkem y" (v tomto pořadí), často jen xRy. Jinak $(x, y) \notin R$, nebo $x \not R\ y$.
(např. $x = y$, $x < y$, $x \not\subseteq y$ místo $(x, y) \in =$, $(x, y) \in <$, $(x, y) \notin \subseteq$)

Příklad

- Relace mezi studenty, kteří získali stejnou známku z DiM.
- Relace mezi dvojicemi studentů, kdo má vyšší skóre z písemky.
- Relace mezi dokumenty s podobnými pojmy (plagiáty).
- Relace mezi záznamy v relační databázi...

(Homogenní) relace na dané množině je speciální případ (heterogenní) relace mezi množinami. Více v předmětu Úvod do logického myšlení.

Definice

(Binární) relace R na množině A je

- reflexivní pokud $(x, x) \in R$ pro všechna $x \in A$,
- symetrická pokud $(x, y) \in R \Leftrightarrow (y, x) \in R$ pro všechna $x, y \in A$,
- antisymetrická pokud $(x, y), (y, x) \in R \Rightarrow x = y$ pro všechna $x, y \in A$,
- tranzitivní pokud $(x, y), (y, z) \in R \Rightarrow (x, z) \in R$ pro všechna $x, y, z \in A$.
- lineární (úplná) pokud $(x, y) \in R$ nebo $(y, x) \in R$ pro každé $x, y \in A$

Příklady

- relace rovnosti „$=$“ je reflexivní, tranzitivní, symetrická i antisymetrická
- relace menší „$<$“ je tranzitivní a antisymetrická, „$\leq$“ je i reflexivní
- relace dělitelnosti „$|$“ na $\mathbb{N}$ (i $\mathbb{N}_0$) je reflexivní, tranzitivní a antisymetrická
- relace „být příbuzný“ je jistě symetrická, tranzitivní a reflexivní
- relace „podřízený/nadřízený“ je antisymetrická a tranzitivní
- relace „dorozumění se“ je obvykle symetrická, nemusí být tranzitivní

Relace

Binární relace R na množině A

Pro konečné a malé $A = \{a_0, a_1, \ldots, a_{n-1}\}$ implementujeme relaci dvourozměrným polem (maticí) `r[ ][ ]`, ve kterém je

`r[i][j] = 0` pokud $(a_i, a_j) \notin R$ a

`r[i][j] = 1` pokud $(a_i, a_j) \in R$.

Příklad

Mějme relaci $R \subseteq [0,4]^2$, kde $R = \{(0,0),(0,4),(1,3)(2,4)\}$.

Relaci R uložíme do dvojrozměrného pole

$$\mathtt{R} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Vlastnosti relací

Mějme binární relaci R na množině $\{0, 1, \ldots, n-1\}$ reprezentovanou dvourozměrným polem `r[][]`.

Test, zda relace `r` je reflexivní, $O(n)$

```
for (i=0; i<n; i++)
   if (!r[i][i]) {    // jsou jedničky?
      printf("Není reflexivní!");
      return -1;
     }
```

Test, zda relace `r` je symetrická, $O(n^2)$

```
for (i=0; i<n; i++)
   for (j=i+1; j<n; j++)
      if (r[i][j]!=r[j][i]) {    // je symetrická matice?
         printf("Není symetrická!");
         return -1;
      }
```

Vlastnosti relací (pokračování)

Test, zda relace `r` je tranzitivní, znamená ověřit pro každou trojici

$$\forall i, j, k: \; r[i][j] \;\wedge\; r[j][k] \;\Rightarrow\; r[i][k].$$

Test, zda je relace `r` tranzitivní, $O(n^3)$

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++) {
      if (!r[i][j]) continue;            // může
      for (k=0; k<n; k++) {
         if (!r[j][k]) continue;         // může
         if (r[i][k]) continue;          // musí být!
         printf("Není tranzitivní!");
         return -1;
      }
   }
```

Příklad

Mějme relaci $R \subseteq [0,4]^2$, kde $R = \{(0,0),(0,4),(1,3)(2,4)\}$ uloženou do dvojrozměrného pole

$$\mathtt{R} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Relace není reflexivní.
Relace není symetrická.
Relace JE tranzitivní.

Otázka

Jak implementovat test antisymetrie?
Jak implementovat test úplnosti?
Jaká je složitost těchto testů?

Permutace

Implementujeme jak bijektivní zobrazení $p : [0, n-1] \to [0, n-1]$.

Příklad

Mějme permutaci $\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 1 & 3 & 0 & 5 \end{pmatrix}$.

Permutaci π uložíme do pole

$$\mathtt{p} = [4\ 2\ 1\ 3\ 0\ 5]\,.$$

Jak poznáme, že v poli je uložena permutace? Stačí ověřit zda p je surjektivní (na).

Test zda `p[ ]` je permutace, $O(n)$

```
for (i=0; i<n; i++)  u[i] = 0;                // pomocné pole
for (i=0; i<n; i++)  if (p[i]>=0 && p[i]<n)  u[p[i]] = 1
   else printf("Není permutace!");           // je mimo
for (i=0; i<n; i++)
   if (u[i]!=1)
      printf("Není permutace!");              // není použit
```

Permutace (pokračování)

Složení permutací `p[ ]` a `q[ ]` je permutace `r[ ]`, $O(n)$

```
for (i=0; i<n; i++)
    r[i] = q[p[i]];
```

Výpis všech cyklů získáme například kódem:

Výpis všech cyklů `n`-prvkové permutace `p[ ]` množiny $[0, n-1]$, $O(n)$

```
for (i=0; i<n; i++)  u[i] = 0;             // pomocné pole
for (i=0; i<n; i++) if (u[i]==0) {         // ještě nepoužit
  printf("(%d",i);  u[i] = 1;              // začátek cyklu
  for (j=p[i]; j!=i; j=p[j])  {            // další v cyklu
    printf(",%d",j);  u[j] = 1;
  }
  printf(")");                             // konec cyklu
}
```

7.2. Implementace množin

Množiny se implementují obtížně. Problémem je

- jejich neuspořádanost (není stanoveno, kam který prvek uložit)
- hlídat, že prvky se nesmí opakovat.

Charakteristická funkce podmnožiny

Musíme znát celé *univerzum* $\mathcal{U} = \{u_0, u_1, \ldots, u_{n-1}\}$, ze kterého vybíráme prvky. Podmnožinu $X \subseteq \mathcal{U}$ implementujeme jako pole `x[ ]`, kde

$$\mathtt{x[i]} = \begin{cases} 1 & \text{pro } u_i \in X \\ 0 & \text{jinak.} \end{cases}$$

Výhody: snadno se hledají prvky množiny, sjednocení je funkcí OR, průnik funkcí AND.
Ověření prvku $\in O(1)$, sjednocení a průnik $O(|\mathcal{U}|)$

Nevýhoda: použitelné jen pro malá univerza $\mathcal{U}$!

Příklad

Množinu $A = \{2, 3, 5\}$ v univerzu $U = [1, 10]$ implementujeme pomocí charakteristické funkce v poli A = [0 1 1 0 1 0 0 0 0 0].

Příklad

Množinu $A = \{2, 3, 5\}$ v univerzu $U = [1, 100]$ implementujeme pomocí charakteristické funkce v poli A = [0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0].

Množinu $B = \{1\}$, B = [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0].

Abychom určili sjednocení $A \cup B$ na U využijeme binární OR.
Pro průnik $A \cap B$ využijeme binární AND.

Příklad

Množinu $A = \{2, 3, 5\}$ v univerzu $U = \mathbb{N}$ implementovat nelze.

Seznam prvků množiny

Množinu X implementujeme seznamem všech prvků. Seznam k prvků množiny X je uložen v poli `x[ ]` a můžeme psát

$$X = \{\mathtt{x}[0], \mathtt{x}[1], \ldots, \mathtt{x}[\mathtt{k}-1]\} \text{ pro pole } \mathtt{x[\ ]} \text{ délky } k.$$

Výhoda: vhodné i pro velmi velká a předem neurčená univerza.

Místo pole je lepší použít *dynamický spojový seznam prvků*, pak lze snadno přidávat a vynechávat prvky v seznamu.

Nevýhoda: vyhledání prvku množiny (nejčastější požadovaná operace) je zdlouhavé – musí se projít celý seznam.

Příklad

Množinu $A = \{2, 3, 5\}$ v univerzu $U = [-MAX\_INT, MAX\_INT]$ implementujeme jako pole A = [2, 3, 5].

Příklad

Množinu $A = \{3, 5, 2\}$ v univerzu $U = [-MAX\_INT, MAX\_INT]$ implementujeme jako pole A = [3, 5, 2].

Ověření, zda prvek `x` v množině `a[ ]` velikosti n, $O(n)$

```
for (i=0; i<n; i++) {              // projdi celé pole a[ ]
   if (a[i]==x) break;             // je x v a[ ]?
}
if (i<n) printf("Prvek x je v poli a[ ]");   // našli?
```

Sjednocení dvou množin daných `a[ ]`,`b[ ]` do pole `c[ ]`, $O(n^2)$

```
for (i=0; i<m; i++)
   c[i] = a[i];                     // z a[ ] všech m prvků
for (i=0,k=m; i<n; i++) {           // b[ ] dalších n prvků
   for (j=0; j<m; j++)
      if (b[i]==a[j]) break;        // je-li b[i] v a[ ]
   if (j<m) continue;               // přeskoč ho
   c[k++] = b[i];                   // jinak ho přidej
}
```

Uspořádaný seznam prvků množiny

Obměna předchozí implementace.
Prvky v seznamu jsou navíc lineárně uspořádány podle předem daného klíče (dle velikosti v případě čísel, dle abecedy u jmen, atd.)

Výhoda: možnost *binárního vyhledávání* prvků v množině metodou půlení intervalu v seznamu (viz příklad).

Příklad

Množinu $A = \{2, 3, 5\}$ v univerzu $U = [-MAX\_INT, MAX\_INT]$ implementujeme jako pole A = [2, 3, 5].

Příklad

Množinu $A = \{3, 5, 2\}$ v univerzu $U = [-MAX\_INT, MAX\_INT]$ implementujeme jako pole A = [2, 3, 5].

Binární vyhledávání čísla k v uspořádaném poli `p[ ]` délky `n`

```
int a = 0; b = n-1;
while (a<b && p[a]!=k) {          // našli?
   c = (a+b)/2;
   if (p[c]<k)  a = c+1;          // ne, bude větší
   else         b = c;            // ne, bude menší
}
if (p[a]!=k)  printf("Číslo k není v seznamu.");
```

Jen $\lceil \log_2 n \rceil$ vyhledávacích kroků.

Přidání každého nového prvku `x` do množiny v poli `a[ ]` vyžaduje $O(n)$ operací:

- nalezení vhodného místa, $O(\lceil \log_2 n \rceil)$
- kopie případně „posunutí“ části pole, $O(n)$

Podobně odebrání prvku.

Sjednocení dvou seřazených množin (polí) `a[ ]`, `b[ ]` velikosti `m`, `n` do seřazeného pole `c[ ]` velikosti `l`, $O(n+m)$

```
int i=0, j=0, k, l=0;
for (k=0; k < m+n; k++) {
   if (i >= m) {                 // je-li a[ ] vyčerpáno
      c[l++] = b[j++];
      continue;
   }
   if (j >= n) {                 // je-li b[ ] vyčerpáno
      c[l++] = a[i++];
      continue;
   }
   if (a[i] == b[j]) {           // průnik jen jednou
      j++;
      continue;
   }
   c[l++] = (a[i] < b[j]) ? a[i++] : b[j++];
}
```

Shrnutí

- Jak velké je univerzum?
- Budeme (a jak často) strukturu množiny modifikovat?
- Budeme (a jak často) v množině vyhledávat?
- Budeme (a jak často) množiny sjednocovat?

… zvolit vhodný model.

7.3. Generování výběrů

Často narazíme na úkol projít všechny výběry některého typu. Všechna

- různá zobrazení,
- variace,
- kombinace bez opakování.

Jednoduché procházení dvojic (trojic, atd. . .)

Všechny uspořádané dvojice indexů `i,j` projdeme dvojí smyčkou

Dvouprvkové variace, $O(n^2)$

```
for (i=0; i<n; i++)                    // dvakrát vnořený cyklus
    for (j=0; j<n; j++) {
       // zpracujeme uspořádanou dvojici (i,j)
    }
```

Všechny neuspořádané dvojice indexů `i,j` projdeme podobně

Dvouprvkové kombinace, $O(n^2)$

```
for (i=0; i<n; i++)                    // jen "nad diagonálou"
   for (j=i+1; j<n; j++) {
      // zpracujeme NEuspořádanou dvojici {i,j}
   }
```

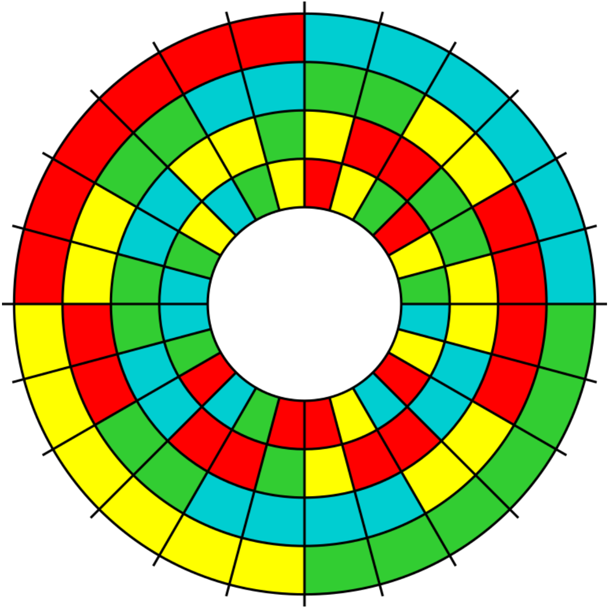
Procházení všech permutací n-prvkové množiny A v poli a[ ]

Všech $n!$ permutací projdeme rekurzivního algoritmu (Heap 1963).

Heapův algoritmus – permutace n prvků

```
int i, a[ ];
permutace(n, a[ ]) {
   if (n==1)
      // zpracujeme permutaci v a[ ]
   else
      for (i=0; i < n-1; i++) {
         permutace(n-1, a[ ]);
         if (n je sudé)
            swap(a[i], a[n-1]);
         else
            swap(a[0], a[n-1]);
      }
      permutace(n-1, a[ ]);
}
```

Funkce `swap(x,y)` jednoduše prohodí obsah proměnných `x` a `y`.



Znázornění kroků Heapova algoritmu.

Procházení všech zobrazení

Všech n^k zobrazení k-prvkové do n-prvkové množiny

$$\mathtt{map} : \{0, 1, \ldots, k-1\} \;\rightarrow\; \{0, 1, \ldots, n-1\}$$

projdeme kódem: Nemusíme sestavovat k vnořených cyklů!

k-prvkové variace s opakováním z n prvků, $O(n^k)$

```
int i, map[k];
map[i = 0] = -1;
while (i>=0) {
   if (++map[i]>=n)              // zvyš o 1
      { i--; continue; }
   if (++i<k)                    // 'nuluj' další prvky
      { map[i] = -1; continue; }
   // zpracujeme zobrazení (map[0], ..., map[k-1])
   i--;
}
```

Pro každou volbu testujeme:

- zda už překročila rozsah `n`, pak se vracíme na předchozí úroveň,
- zda už byla poslední volba `k`-tého prvku, jinak na další úroveň.

Procházení všech k-prvkových variací (bez opakování) z n prvků

k-prvkové variace bez opakování z n prvků, $O(n^k)$

```
int i, j, arrange[k];
arrange[i = 0] = -1;
while (i>=0) {
   if (++arrange[i]>=n)         // zvyš o 1
      { i--; continue; }
   for (j=0; j<i; j++)          // opakuje se?
      if (arrange[i]==arrange[j])  break;
   if (j<i)  continue;          // opakované vynech
   if (++i<k)                   // 'nuluj' další prvky
      { arrange[i] = -1; continue; }
   // zpracujeme variaci (arrange[0],...,arrange[k-1])
   i--;
}
```

Pro každou volbu testujeme:

- zda už překročila rozsah n, pak se vracíme na předchozí úroveň,
- zda se neopakuje předchozí prvek volby, jinak výběr přeskočíme,
- zda už byla poslední volba k-tého prvku, jinak na další úroveň.

Procházení všech kombinací

Projdeme všechny k-prvkové kombinace (bez opakování) z n prvků.
Podobné předchozímu příkladu, avšak nyní všechny výběru generujeme seřazené podle velikosti. Proto každou kombinaci obdržíme jen jednou.

k-prvkové kombinace (bez opakování) z n prvků, $O(n^k)$

```
int i, select[k];
select[i = 0] = -1;
while (i>=0) {
   if (++select[i]>=n)              // zvyš o 1
     { i--; continue; }
   if (++i<k) {
     select[i] = select[i-1];       // jen seřazené!
     continue;
   }
   // zpracujeme kombinaci (select[0],...,select[k-1])
   i--;
}
```

Není třeba testovat opakované výběry, protože prvky kombinace jsou seřazené.

7.4. Generátory náhodných čísel

Skutečně náhodná posloupnosti bitů v počítači.

Kde všude potřebujeme používat náhodná čísla/bity?

- Vytváření náhodných (velkých) privátních klíčů (v SSL certifikátech). Použití náhodného hesla při šifrování SSL přenosu (zde musí být heslo skutečně náhodné, jinak by mohlo být uhodnuto!).
- Řešení kolizí paketů na Ethernetu náhodně dlouhým čekáním s příštím vysíláním.
- Využití při *pravděpodobnostních algoritmech*, využívají náhodných bitů pro výrazné statistické urychlení běhu výpočtu.
- Při statistické analýza algoritmů a reálných procesů, různé modelování chaotických fyzikálních dějů, atd.

Typy náhodných generátorů

Primitivní pseudonáhodné generátory

Využívají aritmetické vzorce typu

$$x := (A \cdot x + B) \mod C.$$

Opakovaně iterujeme a vybrané bity x vytvářejí posloupnost.

Nevýhody: *velká závislost* bitů na předchozích iteracích a *snadná předvídatelnost*.

Pseudonáhodné generátory s vnějším vstupem

Podobné vzorce jako předchozí typ, navíc přidávají vstupy z *vnějších fyzických procesů* (prodlevy stisků kláves, zpoždění operací disku, statistika síťových paketů, atd.)

Problémy: závislost na vnějších podmínkách, *možnost ovlivnění* vnějšími podmínkami, velká „cena" každého bitu.

Hardwarové náhodné generátory

Založeny na různých *kvantových šumech* (třeba přechodové stavy polovodičů).

Problémy: převod šumu na posloupnost uniformních bitů, důvěra v kvantovou mechaniku.

7.5. Kombinatorická exploze

Při programátorském řešení problémů spjatých s diskrétní matematikou často používáme algoritmy typu:

Projdi všechny možnosti a podle nich rozhodni.

Často však narážíme na fenomén zvaný *exponenciální kombinatorická exploze*.

- rychlost růstu funkce faktoriál.
- z Indie: příběh se zrnky obilí na polích šachovnice

Pokud počet prohledávaných možností roste exponenciálně, tak i při zvětšení vstupu o 1 se potřebný čas výpočtu zmnohonásobí.
Stává se, že výpočet pro vstup o velikosti 10 zvládne stroj s procesorem 386, ale vstup o velikosti 15 nelze spočítat ani na nejvýkonnějších počítačích.

Mějte tento fenomén na paměti, až budete programovat procházení možností „hrubou silou“!

Vhodnou volbou algoritmu, datové struktury či omezujících podmínek můžeme dosáhnout výrazného zrychlení výpočtu.

Příklad

Počet různých (neisomorfních!) turnajů pro n týmů (nerozlišujeme pořadí kol ani losování týmů).

n=2 1 turnaj

n=4 1 turnaj

n=6 1 turnaj

n=8 6 turnajů

n=10 396 turnajů

n=12 526 915 620 turnajů

n=14 1 132 835 421 602 062 347 turnajů

n=16 ?

Pokud rozlišujeme losování týmů, tak pro $n = 14$ je 98 758 655 816 833 727 741 338 583 040 turnajů.

Příště

Část II Úvod do Teorie Grafů

Kapitola 1. Pojem grafu

- motivace
- definice grafu
- stupeň vrcholu v grafu