# Matematická olympiáda – kategorie P

# Co se dozvíte?

- Základní informace o MO-P.
- Jak by mělo vypadat řešení?
- Pár užitečných algoritmů.

# Co je MO-P?

- Programovací soutěž,
- zaměřená na efektivní řešení konkrétních úloh
- matematického/teoretického typu;
- vstup a výstup v jednoduchém formátu, žádná další interakce s uživatelem.

Více informací: mo.mff.cuni.cz.

# Organizace soutěže (1)

Domácí kolo:

- Do 15. listopadu.
- 2 teoretické a 2 praktické úlohy.
- Odevzdávání viz mo.mff.cuni.cz.
- Typická postupová hranice: Cca 10 bodů (z 40).

Krajské kolo:

- 23. ledna.
- 4 hodiny, 4 teoretické úlohy řešené na papíře.

# Organizace soutěže (2)

Ústřední kolo:

- 30 nejlepších účastníků.
- 2 $\times$ 5 hodin, 3 teoretické a 3 praktické úlohy
- 20.–22. března v Českých Budějovicích

Výběrové/přípravné soustředění:

- pro cca 10 nejlepších účastníků

# Mezinárodní soutěže

Mezinárodní olympiáda v programování (IOI)

- 4 nejlepší účastníci
- 1.–8.9., Egypt

Středoevropská olympiáda v programování (CEOI)

- 4 nejlepší nematuranti, kteří se nekvalifikovali na IOI
- Brno

# Co by mělo obsahovat řešení – praktické úlohy

- Zdrojový kód řešení:
  - Pascal, C, C++, Java, Python

Vyhodnocení:

- Automatické testování na konkrétních vstupech různé velikosti.
- Musí doběhnout dostatečně rychle a vrátit správnou odpověď.

| body | $n$ |
|---|---|
| 2 | $n \leq 10$ |
| 2 | $n \leq 1\,000$ |
| 3 | $n \leq 100\,000$ |
| 3 | $n \leq 1\,000\,000$ |

# Co by mělo obsahovat řešení – teoretické úlohy

- Co nejsrozumitelnější popis řešení.
- Zdůvodnění jeho správnosti.
- Určení a zdůvodnění časové a paměťové složitosti.
- Zdrojový kód řešení:
  - v libovolném programovacím jazyce nebo pseudokódu
  - je možné vynechat nedůležité části (vstup, výstup, standardní algoritmy)

Vyhodnocení:

- Chybné/neoptimální řešení: 0 bodů.
- Jinak:
  - Cca 3 body za kvalitu popisu a zdůvodnění.
  - 0 – 7 bodů dle efektivity.

# Model

Poslední teoretická úloha:

- delší studijní text
- netradiční výpočetní prostředky
- platí vše co pro ostatní teoretické úlohy
    - „časová složitost" a „zdrojový kód" mohou mít jiný význam, popsaný ve studijním textu

# Příklad

Honza honí yetiho. Ke každému z $n$ předchozích dní si našel novinový článek popisující, kde byl yeti spatřen, a snaží se podle nich určit, jak se pohyboval. Má ale podezření, že informace ve článcích nejsou úplně spolehlivé a občas si někdo vymýšlí. Chtěl by proto pro každých $k$ po sobě jdoucích dnů určit, kde byl dle článků yeti viděn nejčastěji – to je asi nejpravděpodobnější poloha yetiho v té době.

# Soutěžní úloha

Honza pro zjednodušení přiřadil každé možné poloze yetiho číslo mezi 1 a $10^9$. Na vstupu dostanete $n$ čísel poloh yetiho v jednotlivých dnech; pro každých $k$ po sobě jdoucích dní vypište číslo polohy, které se během nich vyskytuje nejčastěji. Jestliže není jednoznačné (tj. stejný maximální počet výskytů v příslušných $k$ dnech má více poloh), vypište 0.

**Příklad vstupu:**

7 3 — $n = 7$ dní, $k = 3$

2 1000 2 3 2 1000 1000 — poloha yetiho

**Odpovídající výstup:**

2 0 2 0 1000

**Omezení na vstupy:**

$$1 \leq k \leq n \leq 10^6$$

# Jednoduché řešení

Pro každý $k$-prvkový úsek vstupu zvlášť určíme, jaké číslo polohy se v něm nejčastěji vyskytuje, a to následujícím postupem:

- Pro každou polohu v aktuálním úseku si spočteme, kolikrát se v něm vyskytuje.
  - Postupně procházíme aktuální úsek a v tabulce `pocty_vyskytu` indexované čísly poloh si počítáme, kolikrát jsme již danou polohu viděli.
  - Tj., pro každé číslo polohy $p$ v aktuálním úseku zvýšíme `pocty_vyskytu[p]` o jedna.
- Vypíšeme polohu $p$, jejíž spočtený počet výskytů je největší možný.

Protože čísla poloh mohou být velká, tabulka `pocty_vyskytu` bude řešena jako hashovací tabulka, která umožňuje v průměrném případě přístup k prvkům v konstantním čase.

# Zdrojový kód

```
unordered_map<int, int> pocty_vyskytu;
for (int z = 0; z <= n - k; z++)  {
  // z je začátek aktuálního úseku
  pocty_vyskytu.clear ();
  for (int i = 0; i < k; i++)
    pocty_vyskytu[poloha[z + i]]++;

  // hledání největšího počtu výskytů
  int max_pocet = 0, max_poloha = 0;
  for (auto &pv : pocty_vyskytu)
    if (pv.second > max_pocet)
      { max_pocet = pv.second;
        max_poloha = pv.first; }
    else if (pv.second == max_pocet)
      max_poloha = 0; // v případě remízy vypišme 0
  printf (" %d", max_poloha);
}
```

# Časová složitost – O-notace

- Určit počet provedených operací přesně je složité.
- Operace trvají různě dlouho, čas jejich provedení závisí na stavu paměti, ...

Místo toho: „Časová složitost je $O(f(n))$.“

- Existuje nějaká konstanta $c$ taková, že pro každé $n \geq 2$ algoritmus provede nejvýše $c \cdot f(n)$ operací.
- Více viz ksp.mff.cuni.cz/kucharky/slozitost/

# Určení časové složitosti

```
for (int z = 0; z <= n - k; z++)  {
  pocty_vyskytu.clear ();
  for (int i = 0; i < k; i++)
    ...

  for (auto &pv : pocty_vyskytu)
    ...
}
```

$$O(n \cdot k)$$

Odhad:

- Cca $10^9$ operací za sekundu, $\approx 100$ operací ve vnitřních cyklech
- Projde pro $n, k \approx \sqrt{10^7} \approx 3000$?

Realita:

- $n = 10000$, $k = 5000$: 0.6 s
- $n = 100000$, $k = 50000$: 100 s

# Běžné složitosti

Současné počítače provádí řádově $10^9$ operací za sekundu:

- $O(n)$ algoritmy bývají efektivní pro cca $n \leq 10^7$
- $O(n^2)$ algoritmus možná pro $n \leq 10\,000$
- $O(2^n)$ pro $n \leq 30$

# Lepší řešení

- Je zbytečné pokaždé znovu přepočítávat celé pole `pocty_vyskytu`!
- Znám: počty výskytů v úseku $z, \ldots, z + k - 1$.
- Chci: počty výskytů v úseku $z + 1, \ldots, z + k$.
  - Stačí odpočítat `poloha[z]`, připočítat `poloha[z+k]`

# Zdrojový kód

```
unordered_map<int, int> pocty_vyskytu;
for (int i = 0; i < k - 1; i++)
  pocty_vyskytu[poloha[i]]++;
for (int z = 0; z <= n - k; z++)  {
  // přidání poslední polohy do aktuálního úseku
  pocty_vyskytu[poloha[z + k - 1]]++;

  // hledání největšího počtu výskytů
  ...

  // odebrání první polohy z aktuálního úseku
  pocty_vyskytu[poloha[z]]--;
}
```

Hledání největšího počtu výskytů stále zabírá čas $O(k)$.

# Vylepšení

- Pro každý počet $m$ $(1, 2, \dots)$ si budu pamatovat seznam poloh, které se v aktuálním úseku vyskytují $m\times$.
- Při zvýšení/snížení počtu výskytů polohu přesunu o jedna výš/níž.
- Nejvyšší neprázdný seznam obsahuje odpověď.
- Musím si pamatovat i místo v seznamu, kde daná poloha je.

# Zdrojový kód

```
typedef list<int>::iterator misto;
unordered_map<int, int> pocty_vyskytu;
unordered_map<int, misto> umisteni;
list<int> krat[k+1];
int max_pocet;

void zvetsi_pocet (int p) {
  int akt = ++pocty_vyskytu[p];
  if (akt > 1)
    krat[akt-1].erase (umisteni[p]);
  umisteni[p] = krat[akt].insert (krat[akt].end (),
                                  p);
  if (akt > max_pocet)
    max_pocet = akt;
}
```

# Zdrojový kód

```
typedef list<int>::iterator misto;
unordered_map<int, int> pocty_vyskytu;
unordered_map<int, misto> umisteni;
list<int> krat[k+1];
int max_pocet;

void sniz_pocet (int p) {
  int akt = --pocty_vyskytu[p];
  krat[akt + 1].erase (umisteni[p]);
  if (akt > 0)
    umisteni[p] =
      krat[akt].insert (krat[akt].end (), p);
  while (krat[max_pocet].empty ())
    max_pocet--;
}
```

# Zdrojový kód

```
for (int i = 0; i < k - 1; i++)
  zvetsi_pocet (poloha[i]);
for (int z = 0; z <= n - k; z++)
  {
    // přidání poslední polohy do aktuálního úseku
    zvetsi_pocet (poloha[z + k - 1]);

    if (krat[max_pocet].size () > 1)
      printf (" 0");
    else
      printf (" %d", krat[max_pocet].back ());

    // odebrání první polohy z aktuálního úseku
    sniz_pocet (poloha[z]);
  }
```

# Časová složitost

```
void zvetsi_pocet (int p) {
  ...
  if (akt > max_pocet)
    max_pocet = akt; // = max_pocet + 1
}

void sniz_pocet (int p) {
  ...
  while (krat[max_pocet].empty ())
    max_pocet--;
}
```

$$O(n)$$

Realita:

- $n = 100000$, $k = 50000$: 0.04 s
- $n = 1000000$, $k = 500000$: 0.6 s

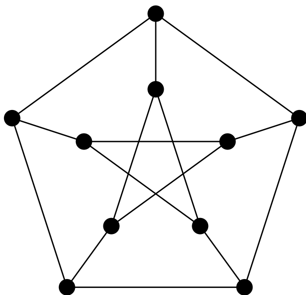
# Co u teoretických úloh nedělat

- Odevzdat pouze zdrojový kód (i komentovaný).
- Přepis programu do češtiny:
  - „Ve for cyklu procházím $z$ od 0 do $n - k$. Pro $i$ od 0 do $k - 1$ k `pocty_vyskytu[poloha[z + i]]` přičtu 1 . . . “
- Nevhodně pojmenované proměnné:
  - „Zavedeme si pole `pole`. Dále si zavedeme pole `pole2`.“
  - „K právě spočítanému číslu přičteme druhý z počtů určených v předminulém odstavci.“
- Pouze popis běhu algoritmu na jednom konkrétním příkladě.
- Rozebírání nedůležitých technických detailů.

# Co naopak dělat

- Popis srozumitelný i bez nahlédnutí do programu.
- Soustředit se na vysvětlení významu, ne na technikálie.
  - „V poli `krat` si na indexu $m$ budeme udržovat seznam poloh, které se v aktuálním úseku vyskytnou právě $m$-krát.“
- Inspirujte se autorskými řešeními.

# Grafy



# Příklady

Dopravní síť ve městě:

- vrcholy: křižovatky
- hrany: ulice

Vztahy:

- vrcholy: lidé
- hrany: znají se?

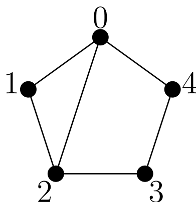
Molekuly:

- vrcholy: atomy
- hrany: vazby mezi nimi

# Reprezentace grafů

Seznam sousedů:

- $0 \to 1, 2, 4$
- $1 \to 0, 2$
- $2 \to 0, 1, 3$
- $3 \to 2, 4$
- $4 \to 0, 3$

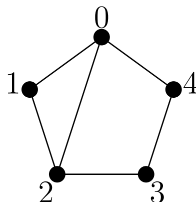



- `vector<list<int>>`
- Stupeň $v$: $\deg v$ = počet sousedů $v$.
- Sousedi vrcholu $v$ v čase $O(\deg v)$.
- Sousedí $u$ s $v$? V čase $O(\min(\deg u, \deg v))$.
- Paměťová složitost: $O(\text{vrcholů} + \text{hran})$.

# Reprezentace grafů

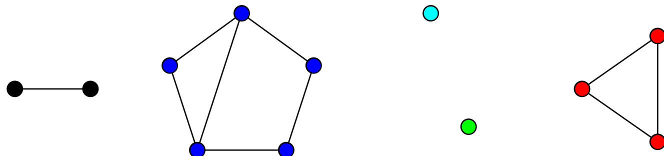
Matice sousednosti:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 0 | 1 |
| 4 | 1 | 0 | 0 | 1 | 0 |




- `bool sousedi[n][n]`
- Sousedi vrcholu $v$ v čase $O(n)$.
- Sousedí $u$ s $v$? V čase $O(1)$.
- Paměťová složitost: $O(n^2)$.

# Komponenty souvislosti



- Graf je souvislý, jestliže se mezi každými dvěma vrcholy dá dostat po hranách.
- Komponenta souvislosti: Maximální souvislá část.

Jak najít komponenty souvislosti?

- Z libovolného vrcholu postupně procházím sousedy, pak sousedy sousedů, . . .
- Dávám si pozor, abych nezpracovával vrchol víc než jednou.

Časová složitost: $O(\text{vrcholů} + \text{hran})$.

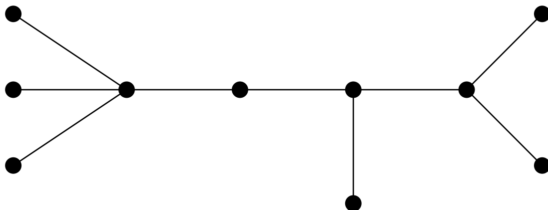
```
vector<list<int>> sousedi;
vector<int> cislo_komponenty (n, -1);
list<int> komponenta (int v, int cislo) {
  list<int> komp{v}; vector<int> fronta{v};
  cislo_komponenty[v] = cislo;
  while (!fronta.empty ()) {
    int x = fronta.back (); fronta.pop_back ();
    for (int y :  sousedi[x])
      if (cislo_komponenty[y] == -1) {
        cislo_komponenty[y] = cislo;
        komp.push_back (y); fronta.push_back (y);
      }}
  return komp;
}
int cislo = 0;
for (int v = 0 ; v < n; v++)
  if (cislo_komponenty[v] == -1)
    komponenta (v, cislo++);
```

# Grafy – více informací

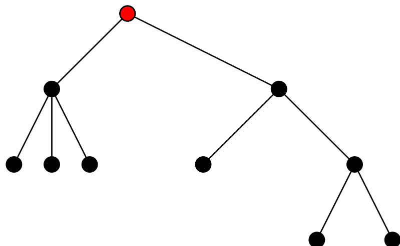
ksp.mff.cuni.cz/kucharky/grafy/

# Stromy

Souvislé grafy bez cyklů.



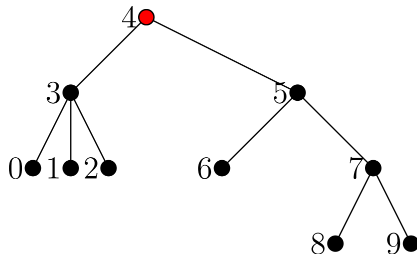
Reprezentace:



- Libovolně zakořeníme.
- Pro každý vrchol:
  - Rodič.
  - Seznam dětí.

# Reprezentace stromu

| | Rodič | Děti |
|---|---|---|
| 0 | 3 | |
| 1 | 3 | |
| 2 | 3 | |
| 3 | 4 | 0,1,2 |
| 4 | $\perp$ | 3,5 |
| 5 | 4 | 6,7 |
| 6 | 5 | |
| 7 | 5 | 8,9 |
| 8 | 7 | |
| 9 | 7 | |




Konstrukce reprezentace:

- Přímo ze zadání (rodič každého vrcholu).
- Průchod podobně jako hledání komponent.

```
vector<list<int>> sousedi;
vector<int> rodic (n, -1);
vector<list<int>> deti (n);
void najdi_deti (int v, int odkud) {
  rodic[v] = odkud;
  for (int y :  sousedi[v])
    if (y != odkud) {
        deti[v].push_back (y);
        najdi_deti (y, v);
      }
}

vector<int> koreny;
for (int v = 0 ; v < n; v++)
  if (rodic[v] == -1) {
      koreny.push_back (v);
      najdi_deti (v, -1);
    }
```

# Další zdroje

- Kuchařky KSP: ksp.mff.cuni.cz/kucharky
- Programátorská liaheň: liahen.ksp.sk
- Průvodce labyrintem algoritmů: pruvodce.ucw.cz
- MO-P FAQ: mo.mff.cuni.cz/p/otazky_a_odpovedi.html