57. ročník Matematické olympiády – 2007/2008

Řešení úloh krajského kola kategorie P

P-II-1 Parkování kočárů

Sluhové musí kočáry zaparkovat tak, aby po svatbě odjela vždy nejprve jedna celá řada kočárů a až poté může začít odjíždět další řada. To znamená, že v každé řadě musí být zaparkovány kočáry, které následují podle důležitosti bezprostředně po sobě. Jinými slovy, jsou-li v řadě za sebou zaparkovány kočáry důležitosti d_1 a d_2, nemůže existovat žádný kočár s důležitostí d_3 takový, že $d_1 < d_3 < d_2$.

Všimněte si, že jakmile zjistíme, že právě parkovaný kočár můžeme postavit do nějaké řady (tzn. jeho zaparkováním do dané řady se neporuší podmínka z prvního odstavce), nic nepokazíme, když ho tam skutečně postavíme.

Na základě těchto úvah můžeme snadno napsat program s časovou složitostí $\mathcal{O}(N^3)$. Budeme si pamatovat všechny dosud vytvořené řady. Když přijede další host, najdeme pro něj vhodnou řadu, nebo vytvoříme novou. Kočár s důležitostí d_i můžeme zaparkovat na konec již existující řady (nechť tato řada končí kočárem s důležitostí d), pokud $d < d_i$ a žádný další host už není mezi nimi, tzn. neplatí $d < d_j < d_i$ pro žádné j. Podobně dokážeme zjistit, zda můžeme přidat kočár na začátek nějaké řady. Jelikož kočárů je N, řad může být v nejhorším případě, jak jsme viděli v posledním příkladu v zadání, až $\Theta(N)$. Protože pro každý kočár a řadu nám test trvá $\mathcal{O}(N)$, máme skutečně kubický algoritmus.

Testování, zda můžeme kočár zaparkovat do dané řady, ovšem dokážeme zrychlit. Stačí si vstup vhodně předzpracovat. Kočáry si přečíslujeme podle jejich důležitosti na čísla $1, 2, \ldots, N$. Potom budou každou řadu tvořit po sobě jdoucí čísla. Řadu kočárů s čísly $k, k+1, \ldots, l$ si zapamatujeme jako dvojici $[k, l]$. Kočár číslo i můžeme přidat na konec řady, která končí kočárem číslo $l = i-1$, nebo na začátek řady, která začíná kočárem číslo $k = i + 1$. Pro každého hosta takto zkontrolujeme maximálně $\mathcal{O}(N)$ řad, takže získáme kvadratický algoritmus, tj. algoritmus s časovou složitostí $\mathcal{O}(N^2)$.

Jakým způsobem kočáry přečíslujeme? Pro každý kočár potřebujeme zjistit jeho pořadí podle důležitosti. Proto všechny kočáry jednoduše setřídíme. Přesněji řečeno, budeme třídit dvojice (d_i, i) podle důležitosti d_i. Po setřídění seznamu dvojic přepíšeme jejich první složky čísly $1, \ldots, N$. Druhé složky nám umožní vrátit kočáry do původního pořadí (stačí dvojice uspořádat podle druhé složky). Třídit můžeme například algoritmem quicksort nebo heapsort.

Naše finální řešení bude (až na přečíslování) lineární a velmi jednoduché. Protože kočáry nyní mají čísla $1, \ldots, N$, stačí si vytvořit jedno velké pole od 0 do $N+1$, v němž si budeme označovat čísla kočárů, které jsme už zaparkovali. Když přijede kočár s číslem i, podíváme se nejprve, zda už jsme zaparkovali kočár číslo $i-1$, tj. zda už je $i-1$ v poli označeno. Pokud ano, můžeme zaparkovat nový kočár na konec nějaké existující řady (za kočár $i-1$). Pokud ne, Zkontrolujeme, zda je v poli označeno $i+1$. Jestliže ano, můžeme kočár umístit na začátek existující řady (před kočár

$i+1$). V opačném případě musíme pro přijíždějící kočár založit novou řadu (zvýšíme počet řad o 1). Kočár číslo i si pak ještě musíme v každém z výše uvedených případů označit v poli. Tento algoritmus má časovou složitost $\mathcal{O}(N)$ plus složitost třídění a paměťovou složitost $\mathcal{O}(N)$.

Jinou možností, jak lze přečíslovat kočáry, je použít asociativní pole (`map` v STL). Stačí kočáry setřídit, do asociativního pole si ke každému zapamatovat pořadí po setřídění, a potom si při parkování kočárů „překládat“ jejich čísla, kdykoliv potřebujeme. Každé takové přeložení stojí čas $\mathcal{O}(\log N)$, což celkově nepřekročí složitost třídění $\mathcal{O}(N \log N)$.

Jiné řešení:

Pro zajímavost uvedeme ještě další řešení, které má ale kvadratickou časovou složitost. Začneme tím, že si zadané pole zkopírujeme a kopii setřídíme, abychom věděli, jaké kočáry se na vstupu nacházejí, a abychom věděli pro každé k, který kočár je k-tý nejmenší v pořadí (stejně jako v prvním řešení úlohy).

V některé řadě musí skončit kočár s nejmenším číslem. Určitě nic nezkazíme, když do této řady umístíme co nejvíce kočárů – ale kolik jich bude?

Dokážeme napsat funkci, která pro dané k ověří, zda můžeme kočáry s k nejmenšími čísly umístit do jedné řady; toto lze udělat právě tehdy, když kočáry s k nejmenšími čísly přijíždějí od nejméně důležitého po nejvíce důležitý. Tuto skutečnost lze snadno ověřit v lineárním čase.

Na tomto principu bude založeno celé řešení: postupným zkoušením zjistíme, kolik nejvíce kočárů můžeme dát do první řady. Potom všechny tyto kočáry vyřadíme a pro zbývající kočáry (pokud ještě nějaké zbyly) provedeme stejný výpočet.

Není příliš těžké ověřit, že časová složitost tohoto řešení je $\mathcal{O}(N^2)$ – „skoro“ při každém volání funkce nám ubude jeden kočár z posloupnosti.

```
{ P-II-1: Parkování kočárů v Pascalu }
const MAXN = 1000000;
var N, K, i, x: longint;
    A: array [1..MAXN] of record       { dvojice, které si pamatujeme }
       a, b: longint;
       end;
    B: array [0..MAXN-1] of boolean;   { které kočáry jsme už zaparkovali }

begin
   { Přečteme vstup a setřídíme kočáry podle důležitosti }
   read(N);
   for i := 1 to N do
      begin
         read(A[i].a);
         A[i].b := i;
      end;
   sort_pairs(A, N);   { funkce na třídění dvojic }

   { Přetřídíme podle původního pořadí }
   for i := 1 to N do
      begin
```

```
         A[i].a := A[i].b;
         A[i].b := i;
      end;
   sort_pairs(A, N);

   { Počítáme řady }
   K := 0;
   for i := 0 to N+1 do
      B[i] := false;
   for i := 1 to N do
      begin
         x := A[i].b;
         if not(B[x-1] or B[x+1]) then
            inc(K);
         B[x] := true;
      end;
   writeln(K);
end.
```

```
// P-II-1: Parkování kočárů v C++
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  int N, K=0;
  vector<pair <int,int> > A;

  scanf ("%d", &N);
  for (int i=0; i<N; ++i) {
    int x; scanf ("%d", &x);
    A.push_back (make_pair (x, i));
  }
  sort (A.begin(), A.end());

  for (int i=0; i<N; ++i) {
    A[i].first = A[i].second; A[i].second = i+1;
  }
  sort (A.begin(), A.end());

  vector<bool> B(N+2, false);
  for (int i=0; i<N; ++i) {
    int x = A[i].second;
    if (!(B[x-1] || B[x+1])) ++K;
    B[x] = true;
  }

  printf ("%d\n", K);
  return 0;
}
```

P-II-2 Dluhopisy

Všimněte si nejprve, že vždy na konci roku (poté, co Kleofáš dostane výnosy) může všechny své dluhopisy prodat a nakoupit nové. Pokaždé by měl nakoupit dluhopisy tak, aby na konci roku dostal co největší výnos. Algoritmus tedy bude vypadat následovně:

1. Na začátku roku nakoupíme co nejvýhodněji dluhopisy.
2. Na konci roku získáme výnosy a prodáme všechny dluhopisy.
3. Pokud chceme ještě aspoň jeden rok pokračovat v investování, pokračujeme opět krokem 1.

Jak výhodně nakupovat?

Pro snazší vyjadřování si nejprve zavedeme následující označení. $V[p]$ bude označovat nejvyšší výnosy, jakých můžeme dosáhnout, máme-li p peněz; c_i bude cena i-tého dluhopisu a v_i je roční výnos i-tého dluhopisu. Pro zadaný obnos K chceme zjistit $V[K]$.

Zjevně nezáleží na tom, v jakém pořadí kupujeme dluhopisy, zajímá nás jen výsledná množina. Některý dluhopis ale musíme koupit jako první. Jakého nejvyššího výnosu můžeme dosáhnout, pokud by to byl dluhopis číslo i? Přeci $v_i + V[K - c_i]$. Koupíme nejprve i-tý dluhopis (výnos z něho bude v_i), a zůstane nám ještě $K - c_i$ peněz. Za ty nakoupíme co nejvýhodněji další dluhopisy.

Jak tedy můžeme zjistit, který dluhopis koupit jako první? Vyzkoušíme všechna možná i a vybereme ten dluhopis, pro nějž bude celkový výnos nejvyšší. Dostáváme tedy vztah:

$$V[K] = \max \{ V[K - c_1] + v_1, V[K - c_2] + v_2, \ldots, V[K - c_D] + v_D \}.$$

Maximum bereme samozřejmě jen přes ty hodnoty, kde $K \geq c_i$, tzn. bereme do úvahy pouze dluhopisy, které za K korun skutečně můžeme nakoupit.

Všimněte si, že k výpočtu $V[K]$ potřebujeme znát hodnoty $V[K - c_1], V[K - c_2], \ldots, V[K - c_D]$. Jak vypočítáme tyto hodnoty? Použijeme postup zvaný dynamické programování – začneme od nejmenších hodnot. Určitě víme, že $V[0] = 0$. Nyní spočítáme $V[1]$. Potom $V[2]$. Takto budeme postupně pokračovat, až se nakonec dostaneme k výpočtu $V[K]$. Každou hodnotu jsme vždy spočítali z předcházejících hodnot, které jsme už v tu chvíli znali.

Všimněte si také, že hodnota K představuje pouze horní hranici, po níž pole vyplňujeme, samotný obsah pole na ní nezávisí. Jinými slovy, když se nám změní finanční situace, nepotřebujeme přepočítávat znovu celé pole V.

Jaký nejvyšší index v poli V nás bude zajímat? Máme dvě možnosti: buď si ho na začátku odhadnout a rovnou spočítat dostatečné množství hodnot, nebo pole V dopočítávat podle potřeby po každém roce.

Rozeberme první zmíněný přístup. V zadání je uvedeno, že výnos z dluhopisu je roven nejvýše 10% ceny dluhopisu. Za rok tedy může Kleofáš zvýšit svůj majetek maximálně o deset procent. Za R roků proto Kleofášův majetek vzroste nejvýše na $K \cdot 1{,}1^R$.

V zadaní je také řečeno, že ceny dluhopisů jsou násobky $T = 1000$. Má-li tedy Kleofáš například 14 947 korun, nemůže si koupit nic jiného než to, co lze nakoupit za 14 000 korun. Proto nám budou stačit hodnoty $V[p]$ pro násobky T. Budeme tedy potřebovat vypočítat $K \cdot 1{,}1^R/T$ hodnot $V[p]$.

Po vypočítání těchto hodnot už jen R-krát zopakujeme postup: „Co nejlépe nakoupit a na konci roku vybrat výnosy“.

Jaká je časová složitost našeho algoritmu? Potřebujeme si spočítat hodnoty $V[p]$, což lze provést v čase $\mathcal{O}(D \cdot K \cdot 1{,}1^R/T)$. Zbytek algoritmu potřebuje čas $\mathcal{O}(R)$, který je zanedbatelný v porovnání s odhadem $\mathcal{O}(D \cdot K \cdot 1{,}1^R/T)$ na celkovou časovou složitost algoritmu.

```
{ P-II-2: Dluhopisy v Pascalu }
const MAXD = 100;
      T = 1000;
      MAXVYNOS = 50000;
var K, D, R, maxV, i, j: longint;
    ceny, vynosy: array [1..MAXD] of longint;
    V: array [0..MAXVYNOS] of longint;

function max(x, y: longint): longint;
begin
   if x>y then max := x
          else max := y;
end;

begin
   { přečteme si vstup }
   read(K, D);
   for i := 1 to D do
      begin
         read(ceny[i], vynosy[i]);
         ceny[i] := ceny[i] div T;
      end;
   read(R);

   { spočítáme všechna V[p] }
   maxV := 2 + round( K*exp(R*ln(1.1)) / T );
   for i := 0 to maxV do
      for j := 1 to D do
         if i-ceny[j] >= 0 then
            V[i] := max(V[i], V[i-ceny[j]] + vynosy[j]);

   { projdeme postupně všechny roky }
   for i := 0 to R-1 do
      K := K + V[K div T];
   writeln(K);
end.

// P-II-2: Dluhopisy v C
#include <stdio.h>
#include <math.h>
```

```
int max(int x, int y)
{
  return (x > y) ? x : y;
}

int main(void) {
    int K, D, R, T = 1000;
    scanf("%d%d", &K, &D);
    int ceny[D], vynosy[D];
    for (int i=0; i<D; i++) {
        scanf("%d%d", &ceny[i], &vynosy[i]);
        ceny[i] /= T;
    }
    scanf("%d", &R);
    int maxV = 2 + (int)( K*pow(1.1,R) / T );
    int V[maxV];
    for (int i=0; i<maxV; i++)
        for (int j=0; j<D; j++)
            if (i-ceny[j] >= 0)
                V[i] = max( V[i], V[i-ceny[j]]+vynosy[j] );
    for (int i=0; i<R; i++)
        K += V[K/T];
    printf("%d\n", K);
    return 0;
}

// P-II-2: Dluhopisy v C++
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;

int main(){
    int K, D, R, T = 1000;
    cin >> K >> D;
    vector<int> ceny(D);
    vector<int> vynosy(D);
    for (int i=0; i<D; i++) {
      cin >> ceny[i] >> vynosy[i];
      ceny[i] /= T;
    }
    cin >> R;
    int maxV = 2 + int( K*pow(1.1,R) / T );
    vector<int> V(maxV);
    for (int i=0; i<maxV; i++)
        for (int j=0; j<D; j++) {
            if (i-ceny[j] < 0) continue;
            V[i] = max( V[i], V[i-ceny[j]]+vynosy[j] );
        }
    for (int i=0; i<R; i++) K+=V[K/T];
    cout << K << endl;
    return 0;
}
```

P-II-3 Piškvorkový turnaj

Základní pozorování

Učiňme nejprve následující pozorování, které bude tvořit základ našeho řešení.

Lemma: Pokud může program x zvítězit v turnaji a program y může porazit program x ve vzájemném zápasu, pak může v turnaji zvítězit i program y.

Nyní právě zformulované Lemma dokážeme. Uvažme turnaj, ve kterém program x zvítězí. Odehrajeme všechny zápasy, ale vynecháme ten, ve kterém je program y vyřazen. Na závěr pak zbydou dva programy, programy x a y, a v posledním zápase program y může vyřadit program x.

Právě dokázané Lemma budeme používat na postupné rozšiřování množiny X programů, které mohou v turnaji zvítězit. Nejprve odsimulujeme libovolný turnaj a tak získáme jeden program x, který patří do množiny X. Pak do množiny X přidáme všechny programy, které mohou program x porazit; poté do množiny X přidáme všechny programy, které mohou porazit nově přidané programy, atd., až už do množiny X není možné přidat žádný další program.

Obsahuje takto vytvořená množina X všechny programy, které mohou v turnaji zvítězit? Označme jako Y množinu programů, které nejsou obsaženy v množině X. Poté, co množinu X již nelze dále rozšířit, všechny programy x z množiny X zvítězí nad libovolným programem y z množiny Y.

Předpokládejme nyní, že existuje program y z množiny Y, který může v turnaji zvítězit. Uvažme turnaj, ve kterém program y zvítězí, a nechť x je program z množiny X, který byl vyřazen jako poslední. Takový program by ale musel být vyřazen nějakým programem z množiny Y (x je poslední program z množiny X v turnaji) a to není možné. Proto program y nemůže v turnaji zvítězit.

Pokud tedy nalezneme algoritmus, který rychle sestrojí množinu X s výše uvedenými vlastnostmi, bude úloha vyřešena.

Algoritmus řešící úlohu

Nejprve odsimulujeme jeden turnaj a vítězný program x vložíme do množiny X. Ty programy, které x vždy porazí, vložíme do množiny Y a zbylé programy vložíme do fronty S.

Dokud se fronta S nevyprázdní, odebereme z S program x, tento program x vložíme do X a všechny programy z Y, které mohou program x porazit, přesuneme z množiny Y do fronty S. Až se fronta S vyprázdní, tak zjevně platí, že libovolný program z množiny X vždy zvítězí nad libovolným programem z množiny Y.

Zbývá tedy vymyslet (rychlou) implementaci výše uvedeného postupu.

Kvadratické řešení

Vytvořit algoritmus s kvadratickou časovou složitostí, tj. složitostí $\mathcal{O}(N^2)$, je jednoduché. Odsimulujeme turnaj, kde v prvním kroku hrají programy 1 a 2, v druhém hraje vítěz prvního souboje s programem 3, v třetím vítěz druhého souboje s programem 4, atd.

V každém kroku algoritmu se množina X zvětší o jeden program, a tedy algoritmus je tvořen nejvýše N kroky. Přesun programů, které může program x porazit, z množiny Y do fronty S lze snadno provést v čase $\mathcal{O}(N)$, a proto celková časová složitost tohoto algoritmu bude $\mathcal{O}(N^2)$.

Lineární řešení

Nejprve si rozmysleme, že pokud máme dvě množiny A a B zadané jako posloupnosti čísel seřazené od nejmenšího po největší, pak lze průnik $A \cap B$ určit v čase $\mathcal{O}(|A| + |B|)$ a tento průnik lze vypsat opět jako posloupnost čísel seřazenou od nejmenšího po největší. Vskutku: porovnejme nejmenší prvky A a B. Pokud jsou stejné, výsledný prvek vložíme do posloupnosti reprezentující množinu $A \cap B$. Pokud jsou různé, odebereme z příslušné množiny menší z těchto dvou prvků. Počet kroků tohoto algoritmu je zjevně omezen $|A \cup B| \leq |A| + |B|$, a tedy jeho časová složitost je $\mathcal{O}(|A| + |B|)$.

Pokud známe jednoho možného vítěze turnaje, kterým je program x, množinu Y si snadno můžeme reprezentovat jako posloupnost $1, \ldots, x-1, x+1, \ldots, N$. Tuto množinu pronikneme výše popsaným postupem s množinou těch programů, které s programem x vždy prohrají. V každém kroku, kdy přesouváme program x z fronty S do množiny X, pronikneme množinu Y s posloupností programů, které program x vždy porazí. Tím se množina Y zmenší na maximálně tolik programů, kolik program x vždy porazí. Pokud i-tý program porazí q_i programů, tak množinu Y pronikáme s množinou velikosti q_x a velikost nově vytvořené množiny Y je nejvýše q_x. Snadno odvodíme, že výpočet nově vzniklých množin Y ve všech krocích algoritmu spotřebuje čas nejvýše $\mathcal{O}(q_1 + \ldots + q_N) = \mathcal{O}(M)$. Celkově tedy bude náš algoritmus vyžadovat čas $\mathcal{O}(M + N)$, pokud zvládneme v tomto čase i odsimulovat nějaký turnaj.

Simulace turnaje

Popišme si nyní, jak lze odsimulovat možný turnaj v čase $\mathcal{O}(M + N)$ pomocí informací zadaných na vstupu. Budeme mít booleovské pole, kde si budeme pro každý program pamatovat, zda je ještě v turnaji. Na začátku vyřadíme všechny programy, nad kterými program 1 určitě vyhraje (to jsou ty, které jsou ve vstupu uvedeny na řádku odpovídajícím programu 1). Na začátku je naším kandidátem na vítěze program $k = 1$. Pomocné pole pak procházíme od začátku a vždy, když narazíme na program p, který je ještě v turnaji, provedeme následující: pokud p vždy zvítězí nad k, položíme $k = p$ a vyřadíme všechny programy, nad kterými p vždy zvítězí. V opačném případě vyřadíme program p z turnaje. Na konci nám zbyde jediný program, program k, který právě odsimulovaný turnaj vyhrál. Zjevně je časová složitost právě popsaného postupu simulace turnaje $\mathcal{O}(M + N)$. Tím jsme dokončili popis lineárního řešení zadané úlohy.

```
{ P-II-3 Piškvorkový turnaj v Pascalu }
const MAXN = 100000;     { Maximální počet programů }
      MAXM = 1000000;    { Maximální počet známých výsledků }
var N: longint;
    { Programy poražené i-tým jsou v W[V[i]..V[i+1]-1] }
```

```
    W: array [1..MAXM] of longint;
    V: array [1..MAXN+1] of longint;
    { Fronta, její čtecí a zapisovací index }
    S: array [1..MAXN] of longint;
    rs, ws: longint;
    { Množiny X a Y a jejich počty prvků; pomocná množina Z }
    X, Y, Z: array [1..MAXN] of longint;
    nx, ny, nz: longint;
    { Pomocné proměnné }
    i, k, a, b: longint;
    tmp: array [1..MAXN] of boolean;

procedure nacti;
var i, j, p, d: longint;
begin
   read(N);
   p := 1;
   for i := 1 to N do
      begin
         read(d);
         V[i] := p;
         for j := 1 to d do
            begin
               read(W[p]);
               inc(p);
            end;
      end;
   V[N+1] := p;
end;

function simuluj: longint;
var hra: array [1..MAXN] of boolean;  { kdo ještě zbývá v turnaji }
    i, kandidat, dalsi: longint;
begin
   for i := 1 to N+1 do
      hra[i] := true;
   kandidat := 1;
   for i := V[kandidat] to V[kandidat+1]-1 do
      hra[W[i]] := false;
   for dalsi := 2 to N do
      if hra[dalsi] then
         begin
            hra[kandidat] := false;
            kandidat := dalsi;
            for i := V[kandidat] to V[kandidat+1]-1 do
               hra[W[i]] := false;
         end;
   simuluj := kandidat;
end;

begin
   nacti;
   k := simuluj;    { najdeme nějakého možného vítěze }

   { inicializujeme množiny X a Y }
```

```
nx := 1;
X[1] := k;
ny := V[k+1] - V[k];
for i := 1 to ny do
   Y[i] := W[V[k]+i-1];

{ do fronty S vložíme všechny prvky, které nejsou v X ani v Y }
rs := 1; ws := 1;
for i := 1 to N do
   tmp[i] := false;
tmp[k] := true;
for i := 1 to ny do
   tmp[Y[i]] := true;
for i := 1 to N do
   if not tmp[i] then
      begin
         S[ws] := i;
         inc(ws);
      end;

{ z fronty S přidáváme do množiny X a upravujeme množinu Y }
while rs < ws do
   begin
     i := S[rs];    { vyjmeme i z fronty S }
     inc(rs);
     inc(nx);       { vložíme ho do X }
     X[nx] := i;
     nz := 0;       { a půjdeme přepočítat Y, zatím do pomocné Z }
     a := 1;
     b := 0;
     while (a <= ny) and (b < V[i+1]-V[i]) do
        begin
          if Y[a] = W[V[i]+b] then
             begin
                inc(nz);
                Z[nz] := Y[a];
                inc(a);
                inc(b);
             end
          else if Y[a] < W[V[i]+b] then
             begin
                S[ws] := Y[a];
                inc(ws);
                inc(a);
             end
          else
             inc(b);
        end;
     for i := 1 to nz do   { Y := Z }
        Y[i] := Z[i];
     ny := nz;
   end;

{ vypíšeme množinu X }
for i := 1 to nx do
```

```
      write(X[i], ' ');
   writeln;
end.
```

```
// P-II-3 Piškvorkový turnaj v C++
#include <algorithm>
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int N; // počet programů
vector< vector<int> > vyhraje; // pro každý program seznam těch, nad kterými vyhraje

void nacti() { // načte vstup
  cin >> N;
  vyhraje.resize(N+1);
  for (int i=1; i<=N; i++) {
    int d;
    cin >> d;
    vyhraje[i].resize(d);
    for (int j=0; j<d; j++) cin >> vyhraje[i][j];
  }
}

int simuluj() {
  vector<bool> hra(N+1,true); // u programu, zda je ještě v turnaji
  int kandidat = 1;
  for (unsigned i=0; i<vyhraje[kandidat].size(); i++)
    hra[ vyhraje[kandidat][i] ] = false;
  for (int dalsi=2; dalsi<=N; dalsi++)
    if (hra[dalsi]) {
      hra[ kandidat ] =0;
      kandidat = dalsi;
      for (unsigned i=0; i<vyhraje[kandidat].size(); i++)
        hra[ vyhraje[kandidat][i] ] = false;
    }
  return kandidat;
}

int main() {
  nacti();
  int v = simuluj(); // najdeme prvního možného vítěze

  // inicializujeme X a Y
  vector<int> X, Y;

  X.push_back(v);
  Y = vyhraje[v];

  // do fronty S vložíme všechny prvky, které nejsou v X ani v Y
  queue<int> S;
  vector<int> tmp(N+1);
  tmp[v] = 1;
  for (unsigned i=0; i<Y.size(); i++) tmp[Y[i]] = 1;
```

```
  for (int i=1; i<=N; i++) if (!tmp[i]) S.push(i);

  // z fronty S přidáváme do množiny X a upravujeme množinu Y
  while (!S.empty()) {
    int x = S.front(); S.pop();
    X.push_back(x);
    // přepočítáme množinu Y
    vector<int> newY;
    unsigned a=0, b=0;
    while (a<Y.size() && b<vyhraje[x].size()) {
      if (Y[a] == vyhraje[x][b]) { newY.push_back(Y[a]); a++; b++; }
      else if (Y[a] < vyhraje[x][b]) { S.push(Y[a]); a++; }
      else if (Y[a] > vyhraje[x][b]) b++;
    }
    Y = newY;
  }
  for (unsigned i=0; i<X.size(); i++) cout << X[i] << ' ';
  cout << endl;
}
```

Jiné lineární řešení

Popišme si nyní krátce jiné možné řešení, jehož časová složitost je také lineární, čili $\mathcal{O}(M+N)$.

Nejdříve vytvoříme posloupnost $p_1, \ldots, p_N$ takovou, že program p_i může vyhrát nad programem p_{i+1}. Předpokládejme, že jsme již vytvořili takovou posloupnost $p_1, \ldots, p_k$ pro programy $1, \ldots, k$. Nalezneme největší index j takový, že program p_j může vyhrát nad programem $k+1$. Program $k+1$ vložíme těsně za program p_j. Pokud program $k+1$ vždy porazí všechny programy $1, \ldots, k$, vložíme program $k+1$ na začátek posloupnosti. Nově vytvořená posloupnost má opět požadované vlastnosti: program p_j může vyhrát nad programem $k+1$ a podle volby indexu j, program p_{j+1} vždy prohraje s programem $k+1$.

Výše uvedený postup lze implementovat v čase $\mathcal{O}(M+N)$ pomocí dvou polí P a B délky N. V poli P si budeme pamatovat aktuální posloupnost. Všimněte si, že při vložení programu k je potřeba posunout nejvýše q_k posledních členů posloupnosti o jednu pozici doprava, abychom mohli prvek k do posloupnosti v poli P vložit.

Druhé pole B na začátku inicializujeme nulami a při zpracovávání programu $k+1$ vložíme hodnotu k na místa $B[i]$ taková, že program $k+1$ vždy porazí program i. Index j pak určíme tak, že vytvořenou posloupnost $p_1, \ldots, p_k$ procházíme od konce, tj. od $j=k$, a index j zmenšujeme tak dlouho, dokud platí $B[p_j] = k+1$. Výsledný index j je největší index takový, že program p_j může porazit program $k+1$. Pokud $j=0$, program $k+1$ vždy porazí všechny programy $1, \ldots, k$. Je snadné si rozmyslet, že i tato fáze zabere čas $\mathcal{O}(q_k)$ a tedy celkový čas na vytvoření posloupnosti $p_1, \ldots, p_N$ je $\mathcal{O}(M+N)$.

Podle Lemmatu, pokud program p_i může zvítězit v turnaji, pak v turnaji mohou zvítězit i programy $p_1, \ldots, p_{i-1}$. Tedy musí existovat index i_0, takový, že v turnaji mohou zvítězit právě programy $p_1, \ldots, p_{i_0}$ a programy $p_{i_0+1}, \ldots, p_N$ nemohou nikdy zvítězit. Zbývá tedy dořešit, jak nalezneme index i_0.

Na začátku položíme $i_0 = 1$ a budeme postupně zpracovávat prvky p_k posloupnosti od $k = 1, \ldots$, dokud $k \leq i_0$. Stejným postupem, jako jsme sestrojili posloupnost $p_1, \ldots, p_N$, určíme největší index j takový, že program p_j může porazit program p_k. Pokud $j > i_0$, zvýšíme hodnotu i_0 na j (což lze podle Lemmatu). Poté zvýšíme hodnotu k o jedna a pokračujeme se zpracováváním dalšího prvku posloupnosti. Z popisu programu je vidět, že všechny programy $p_{i_0+1}, \ldots, p_N$ vždy prohrají se všemi programy $p_1, \ldots, p_{i_0}$ a programy $p_1, \ldots, p_{i_0}$ mohou v turnaji zvítězit. Tedy programy $p_1, \ldots, p_{i_0}$ tvoří množinu X a programy $p_{i_0+1}, \ldots, p_N$ tvoří množinu Y; množiny X a Y pak mají vlastnosti uvedené na začátku řešení této úlohy.

Celková časová složitost právě popsaného řešení je tedy také $\mathcal{O}(M + N)$.

Další možná řešení

Zmiňme se stručně i o jiných možných (byť méně efektivních) řešeních úlohy. Uvažme orientovaný graf, jehož vrcholy reprezentují programy $p_1, \ldots, p_N$ a graf obsahuje hranu z programu p_i do programu p_j, pokud program p_i může vyhrát nad programem p_j, tj. p_j vždy nevyhraje nad programem p_i. Podobně jako v ostatních řešeních nejdříve odsimulujeme turnaj a získáme jeden program x, který může turnaj vyhrát. Podle Lemmatu pak v turnaji mohou zvítězit právě ty programy, ze kterých vede orientovaná cesta do vrcholu reprezentujícího program x. Prohledání orientovaného grafu z vrcholu x nám poskytne řešení, jehož časová složitost je $\mathcal{O}(N^2)$.

Konečně řešení s časovou složitostí $\mathcal{O}(N^3)$ lze získat i tak, že si uvědomíme, že program x může v turnaji zvítězit právě tehdy, když z něj vede orientovaná cesta do každého dalšího vrcholu v orientovaném grafu. Otestovat, zda z jednoho vrcholu vede orientovaná cesta do všech ostatních, lze v čase $\mathcal{O}(N^2)$, a tedy celková časová složitost tohoto řešení bude $\mathcal{O}(N^3)$.

P-II-4 Překládací stroje

Podúloha a)

Budeme postupovat podobně jako v domácím kole. Jako první sestrojíme překládací stroj, který zkopíruje vstupní řetězec na výstupní a navíc do něj může přidat libovolné množství písmen c. Tento stroj může vypadat například následovně:

$$\begin{aligned} Z_1 &= (K_1, \Sigma, P_1, \spadesuit, F_1) \\ \Sigma &= \{a, b, c\} \\ K_1 &= \{\spadesuit\} \\ F_1 &= \{\spadesuit\} \\ P_1 &= \{(\spadesuit, a, a, \spadesuit),\ (\spadesuit, b, b, \spadesuit), (\spadesuit, \varepsilon, c, \spadesuit)\}. \end{aligned}$$

Množina $M_2 = Z_1(M_1)$ tedy obsahuje právě ty řetězce písmen a, b a c, které obsahují stejný počet písmen a a b.

Analogicky lze zkonstruovat druhý stroj, který množinu M_1 přeloží na množinu řetězců se stejným počtem písmen b a c a libovolným počtem písmen a. Pro změnu

však takovouto množinu řetězců vyrobíme z množiny M_2 tak, že písmena a zaměníme za písmena b, písmena b za písmena c a písmena c za písmena a. Překládací stroj s těmito vlastnostmi je následující:

$$\begin{aligned} Z_2 &= (K_2, \Sigma, P_2, \clubsuit, F_2) \\ \Sigma &= \{a, b, c\} \\ K_2 &= \{\clubsuit\} \\ F_2 &= \{\clubsuit\} \\ P_2 &= \{(\clubsuit, a, b, \clubsuit),\ (\clubsuit, b, c, \clubsuit), (\clubsuit, c, a, \clubsuit)\}. \end{aligned}$$

Sestrojili jsme tedy nyní množinu $M_3 = Z_2(M_2)$, která obsahuje právě ty řetězce písmen a, b a c se stejným počtem písmen b a c.

Je zřejmé, že konstruovanou množinu řetězců získáme průnikem těchto množin M_2 a M_3.

Podúloha b)

Hlavním trikem při řešení této podúlohy je uvědomit si, že pokud je výstupní množina řetězců dostatečně jednoduchá, pak nemusíme vstupní množinu řetězců vůbec použít, neboť výstupní množinu zvládneme vygenerovat i bez jejího použití. V této podúloze bude tedy výsledný překládací stroj pracovat následovně:

1. Stroj nejdříve přečte celý vstupní řetězec a „zahodí“ ho (podle definice překládacího stroje je nutné přečíst celý vstupní řetězec).
2. Stroj poté začne generovat číslo a ve stavu překládacího stroje si pamatujeme zbytek dosud vypsaného čísla po dělení sedmi. Koncový stav bude odpovídat číslům se zbytkem 0 po dělení sedmi (právě tehdy totiž lze ukončit generování zápisu čísla).

Ještě je třeba dát pozor, aby číslo, které vygenerujeme, nezačínalo nulou, ale toto snadno zajistíme při přechodu ze stavu, kdy čteme vstupní řetězec, do stavů, kdy generujeme výstupní řetězec. Výsledný překládací stroj může tedy vypadat takto:

$$\begin{aligned} Z &= (K, \Sigma, P, \textbf{čti}, F) \\ \Sigma &= \{0, 1, \ldots, 9\} \\ K &= \{\textbf{čti}\} \ \cup\ \{0, 1, \ldots, 6\} \\ F &= \{0\} \\ P &= \big\{(\textbf{čti}, x, \varepsilon, \textbf{čti}) \ \big|\ x \in \{0, \ldots, 9\}\big\} \cup \\ &\quad \cup \big\{(\textbf{čti}, \$, y, y \bmod 7) \ \big|\ y \in \{1, \ldots, 9\}\big\} \cup \\ &\quad \cup \big\{(x, \varepsilon, y, (10x + y) \bmod 7) \ \big|\ x \in \{0, \ldots, 6\},\ y \in \{0, \ldots, 9\}\big\}. \end{aligned}$$

Popišme si nyní práci tohoto překládacího stroje slovně: stroj setrvá ve stavu **čti**, dokud nepřečte celý vstupní řetězec, a pak při přechodu do jednoho ze stavů $0, \ldots, 6$ vypíše nenulovou číslici $1, \ldots, 9$ na začátek výstupního řetězce. Stavy $x = 0, \ldots, 6$ reprezentují, že dosud vypsaná část řetězce dává po dělení sedmi zbytek x. Za dosud

vygenerovaný vstupní řetězec lze vypsat libovolnou číslici $y = 0, \ldots, 9$ a přejít do stavu $(10x + y) \bmod 7$, který bude udávat zbytek po dělení sedmi dosud vygenerovaného výstupního řetězce. Generování výstupního řetězce můžeme ukončit, kdykoliv jsme ve stavu 0 (tehdy dosud vygenerovaný řetězec reprezentuje číslo dělitelné sedmi). Vidíme, že překladem vstupní množiny vznikne množina desítkových zápisů kladných čísel dělitelných sedmi.