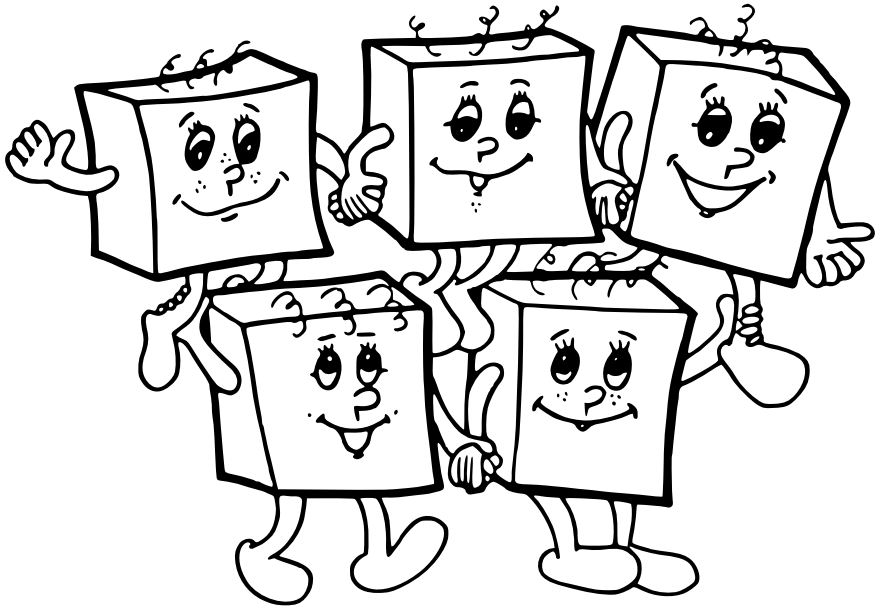
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH



dvadsiaty tretí ročník

školský rok 2007/08

riešenia krajského kola
kategória A

- **Olympiáda v informatike** je od školského roku 2006/07 samostatnou súťažou. Predchádzajúcich 21 ročníkov tejto súťaže prebiehalo pod názvom **Matematická olympiáda, kategória P** (programovanie).
- Oficiálnu **webstránku** súťaže nájdete na `http://www.ksp.sk/oi/`.

A-II-1 Parkovanie kočov

Sluhovia majú koče zaparkovať tak, aby pri odchode vždy najskôr odišiel jeden celý rad, až potom začal odchádzať ďalší. To znamená, že v každom rade musia byť zaparkované koče, ktoré idú podľa dôležitosti bezprostredne po sebe. Inými slovami, ak sú v rade za sebou zaparkované koče dôležitosti d_1 a d_2, nemôže existovať koč s dôležitosťou d_3 taký, že $d_1 < d_3 < d_2$.

Všimnite si tiež, že akonáhle zistíme, že koč, ktorý práve parkujeme, môžeme postaviť do nejakého radu, nič nepokazíme, ak ho tam naozaj postavíme.

Na základe týchto pozorovaní sa dá napísať program s časovou zložitosťou $O(N^3)$. Budeme si pamätať všetky dosiaľ vytvorené rady. Keď príde ďalší hosť, nájdeme pre neho vhodný rad, alebo vytvoríme nový. Koč s dôležitosťou d_i môžeme zaparkovať na koniec už existujúceho radu (nech sa tento rad končí kočom s dôležitosťou d), ak $d < d_i$ a žiadny ďalší hosť už nie je medzi nimi, teda neplatí $d < d_j < d_i$ pre žiadne j. Podobne vieme zistiť, či koč môžeme pridať na začiatok nejakého radu. Keďže kočov je N, radov môže byť v najhoršom prípade, ako sme videli v poslednom príklade v zadaní, až $\Theta(N)$ a keďže pre každý koč a rad nám test trvá $O(N)$, máme naozaj kubický algoritmus.

Testovanie, či koč môžeme zaparkovať do daného radu, vieme zrýchliť: Stačí si vstup predspracovať. Koče prečíslujeme (stále podľa ich dôležitosti) na čísla $1, 2, \ldots, N$. Potom budú rad tvoriť po sebe idúce čísla. Rad kočov s číslami $k, k+1, \ldots, l$ si stačí pamätať ako dvojicu $[k, l]$. Koč číslo i môžeme pridať na koniec radu, ktorý končil kočom číslo $l = i - 1$, alebo na začiatok radu, ktorý začínal kočom číslo $k = i + 1$. Pre každého hosťa prezrieme maximálne $O(N)$ radov, takže máme kvadratický algoritmus.

Ako koče prečíslujeme? Pre každý koč potrebujeme zistiť jeho poradie podľa dôležitosti. Preto koče jednoducho utriedime. Presnejšie, budeme triediť dvojice (d_i, i) podľa dôležitosti d_i. Po zotriedení prvé zložky prepíšeme číslami $1, \ldots, N$. Druhé zložky nám pomôžu vrátiť koče do pôvodného poradia (stačí dvojice usporiadať podľa druhej zložky). Triediť môžeme napríklad quick-sortom alebo heap-sortom.

Vzorové riešenie je (až na prečíslovanie) lineárne a veľmi jednoduché. Keďže koče majú teraz čísla $1, \ldots, N$, stačí si spraviť jedno veľké pole od 0 po $N + 1$, kde budeme zaškrtávať čísla kočov, ktoré sme už zaparkovali. Keď príde koč s

číslom i, pozrieme sa, či už sme zaparkovali koč číslo $i-1$ (či je $i-1$ odškrtnuté; ak áno, môžeme ho dať na koniec existujúceho radu), alebo či je odškrtnuté $i+1$ (ak áno, môžeme koč dať na začiatok existujúceho radu). V opačnom prípade musíme vytvoriť nový rad (zvýšime počet radov o 1). Koč číslo i odšktrneme. Takýto algoritmus má časovú zložitosť $O(N)$ plus zložitosť triedenia a pamäťovú zložitosť $O(N)$.

Iná možnosť ako prečíslovať koče je použiť asociatívne pole (`map` v STL). Stačí koče utriediť, do asociatívneho poľa si ku každému zapamätať poradie po utriedení, a potom pri parkovaní kočov si „prekladať“ ich čísla keď potrebujeme.

Iné riešenie

Pre zaujímavosť uvedieme ešte jedno trochu iné riešenie. Začneme tým, že si zadané pole skopírujeme a kópiu utriedime, aby sme vedeli, aké koče sa na vstupe nachádzajú, a aby sme vedeli pre každé k, ktorý koč je k-ty najmenší v poradí.

V niektorom rade musí skončiť najmenší koč. Určite nič nepokazíme, ak do tohto radu umiestnime čo najviac kočov – ale koľko to bude?

Vieme napísať funkciu, ktorá pre dané k overí, či vieme k najmenších kočov umiestniť do jedného radu – simulujeme umiestňovanie, pričom koče väčšie ako k-ty najmenší ignorujeme.

Takto vieme spraviť riešenie: postupným skúšaním zistíme, koľko najviac kočov vieme dať do prvého radu. Potom všetky tieto koče vyhodíme z poradia a pre zvyšné koče (ak ešte nejaké zostali) začneme úplne odznova ten istý proces.

Nie je to na prvý pohľad evidentné, ale časová zložitosť takéhoto riešenia je $O(N^2)$. (Myšlienka dôkazu: za skoro každé zavolanie overovacej funkcie nám pribudne jeden koč, ktorý už vieme zaparkovať.)

Listing programu:

```
#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);
}
```

A-II-2 Dlhopisy

Prvá vec, ktorú sme si mohli všimnúť je to, že na konci roka (po tom, ako Kleofáš dostane výnosy) vždy môže všetky dlhopisy predať a nakúpiť nové. Vždy by ich mal nakúpiť tak, aby na konci roka dostal čo najvačšie výnosy. Algoritmus bude vyzerať nasledovne:

1. Na začiatku roka nakúp čo najlepšie.
2. Na konci roka vyber výnosy a predaj všetky dlhopisy.
3. Ak chcem ešte jeden rok pokračovať, tak choď na krok 1.

Ako výhodne nakupovať?

Pre jednoduchosť vyjadrovania si najkôr zaveďme nejaké označenia. $V[i]$ odteraz označuje najväčšie výnosy, ktoré vieme dosiahnuť ak máme i peňazí. c_i bude cena i-teho dlhopisu a v_i je ročný výnos i-teho dlhopisu. Keď dostaneme obnos peňazí K, tak chceme zistiť $V[K]$.

Zjavne je jedno v akom poradí kupujeme dlhopisy, zaujíma nás len výsledná množina. Niektorý dlhopis ale musíme kúpiť ako prvý. Aké vieme mať najvyššie výnosy, ak by to bol dlhopis číslo i? Predsa $v_i + V[K - c_i]$. Ľudovou rečou

povedané, kúpime najskôr i-ty dlhopis (výnos z neho bude v_i), a ostane nám $K - c_i$ peňazí. Za tie nakúpime čo najvýhodnejšie.

Ako teda zistiť, ktorý dlhopis kúpiť ako prvý? Vyskúšame všetky možné i a vyberieme ten, pre ktorý bude celkový výnos najväčší. Teda dostávame vzťah:

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

(Samozrejme, maximum berieme len cez tie hodnoty, kde $K \geq c_i$, teda berieme do úvahy len tie dlhopisy, ktoré za K peňazí môžeme kúpiť.)

Všimnite si, že na vypočítanie $V[K]$ potrebujeme poznať $V[K - c_1], V[K - c_2], \cdots, V[K - c_D]$. Ako vypočítame tie? Na to si pomôžeme postupom, ktorý sa volá dynamické programovanie – začneme od najmenších hodnôt. Určite vieme, že $V[0] = 0$. Teraz si vypočítame $V[1]$. Potom $V[2]$. Takto budeme postupne pokračovať, až nakoniec dostaneme k výpočtu $V[K]$. Každú hodnotu sme vypočítali z predchádzajúch hodnôt, ktoré sme už tou dobou poznali. A teda aj v čase, keď sme sa dostali k výpočtu $V[K]$, sme už mali vypočítané všetky potrebné hodnoty.

Tiež si môžeme všimnúť, že hodnota K predstavuje len hornú hranicu, po ktorú pole vypĺňame. Samotný obsah poľa od nej nezávisí. Inými slovami, keď sa nám zmení finančná situácia, nepotrebujeme preratúvať celé pole V odznova.

Aký najväčší index v poli V nás bude zaujímať? Máme dve rovnocenné možnosti: buď si ho na začiatku odhadnúť a rovno spočítať dosť veľa hodnôt, alebo pole V dopočítavať podľa potreby po každom roku.

Prvý prístup: V zadaní sa dalo dočítať, že výnos z dlhopisu je najviac 10% z ceny dlhopisu. Teda za rok vie Kleofáš znásobiť svoj majetok maximálne o desať percent. Za R rokov teda Kleofášov majetok narastie nanajvýš na $K \cdot 1.1^R$.

V zadaní sa tiež môžeme dočítať, že ceny dlhopisov sú násobky $T = 1000$. Teda ak napr. má Kleofáš 14 947 korún, nemôže si nakúpiť nič iné ako to, čo zvládne nakúpiť za 14 000 korún. Preto nám budú stačiť hodnoty $V[i]$ pre násobky T. Budeme teda potrebovať vypočítať $K \cdot 1.1^R/T$ hodnôt $V[i]$.

Po vypočítaní týchto hodnôt už len R krát zopakujeme postup: „čo najlepšie nakúpiť a na konci roka vybrať výnosy“.

Aká je časová zložitosť nášho algoritmu? Potrebujeme si predrátať hodnoty

$V[i]$, čo vieme spraviť v čase $O(D \cdot K \cdot 1.1^R/T)$. Zvyšok algoritmu beží v zanedbateľnom čase $O(R)$.

Listing programu:

```
#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), 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)
                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;
}
```

A-II-3 O víťazovi turnaja

Základné pozorovanie

Začneme základným pozorovaním, ktoré povedie k efektívnemu riešeniu úlohy:

♡: Ak program x môže vyhrať turnaj, a program y môže vyhrať v zápase s programom x, tak aj program y môže vyhrať celý turnaj.

Toto pozorovanie si teraz dokážeme. Zoberme si nejaké poradie zápasov a výsledkov, ktoré vedie k tomu, že x vyhrá turnaj. V práve jednom z týchto zápasov y prehrá a z turnaja vypadne. Predstavme si teraz, že odohráme všetky zápasy presne rovnako, až na to, že vynecháme zápas, v ktorom by y vypadol. Dostaneme poradie zápasov, ktoré keď sa zahrajú, ostanú v turnaji len dvaja

hráči – x a y. No a keďže y môže vyhrať nad x, stačí teraz pridať na koniec zápas, v ktorom y porazí x.

Naše pozorovanie ♡ vieme teda použiť na postupné zisťovanie ďalších a ďalších programov, ktoré turnaj vyhrajú. Potrebujeme ale niekde začať – vedieť aspoň jeden program, ktorý môže turnaj vyhrať. Ako ho zistiť? To je ľahké – stačí jednoducho jeden možný turnaj odsimulovať a zobrať jeho víťaza.

Máme teda už prvú predstavu, ako môže naše vzorové riešenie vyzerať:

1. Odsimuluj jeden turnaj a nájdi prvého možného víťaza.
2. Používaním ♡ pridávaj nových možných víťazov.

Nájdeme všetkých víťazov?

Z toho, čo sme zatiaľ povedali, ešte nevyplýva odpoveď na jednu dôležitú otázku: Nájdeme naším postupom naozaj **všetky** programy, ktoré môžu turnaj vyhrať?

Zjavne ♡ môžeme použiť len konečne veľa krát. Skôr či neskôr sa dostaneme do situácie, že už použitím ♡ nevieme o žiadnom ďalšom programe povedať, že môže vyhrať turnaj.

V tomto okamihu máme programy rozdelené do dvoch množín. V prvej (nazvime ju V ako víťazi) sú programy, o ktorých už vieme, že môžu turnaj vyhrať. V druhej (tú nazveme P ako porazení) sú ostatné programy.

Keďže použitím ♡ nevieme žiaden program presunúť z P do V, znamená to, že každý program z V nutne vyhrá nad každým programom z P.

Teraz tvrdíme, že žiaden program, ktorý je v tomto okamihu v P, turnaj nemôže vyhrať.

Prečo je to tak? Všimnime si priebeh ľubovoľného turnaja. Skôr či neskôr ostane v turnaji z programov z množiny V už len jeden. No a ten už nemá ako vypadnúť – všetky ostatné programy, ktoré sú ešte v turnaji, s ním totiž prehrajú. Preto tento program nutne turnaj vyhrá. Víťaz je teda vždy z nami zostrojenej množiny V.

Dokázali sme teda, že algoritmus „kým sa dá použiť ♡, pridávaj do V nové programy“ naozaj nájde všetky programy, ktoré môžu vyhrať turnaj. Zostáva zodpovedať otázku, ako efektívne vieme tento algoritmus implementovať.

Podrobnejší popis algoritmu

Prvým krokom nášho programu bude simulácia jedného turnaja. Tým dostaneme jedného víťaza v.

Počas zvyšku algoritmu budeme postupne zostojovať množinu víťazov V. Zároveň s tým si budeme udržiavať aj množinu P tých programov, ktoré s každým programom z aktuálnej množiny V nutne prehrajú.

Všimnime si, že zatiaľ čo množinu V budeme zväčšovať, množina P sa nikdy zväčšiť nemôže. Keď pridáme nový program do V, jediné, čo sa stane, je, že z P ubudnú tie programy, ktoré nad ním môžu vyhrať.

Tiež si všimnime, že až kým náš algoritmus neskončí, budú existovať programy, ktoré nie sú ani vo V, ani v P. O týchto už vieme, že môžu vyhrať nad niektorým programom z V, ale ešte sme ich do V nepridali. Aby sme ich nemuseli zakaždým hľadať, budeme si tieto programy čakajúce na spracovanie pamätať vo fronte S.

Keď teda zistíme prvého možného víťaza, nastavíme si $V = \{v\}$. Ďalej P bude množina programov, s ktorými v určite vyhrá. No a všetky ostatné programy vložíme do fronty S. Kým fronta S nie je prázdna, dokola opakujeme:

1. Vyberieme program x z fronty S.
2. Pridáme program x do množiny V.
3. Prejdeme prvky v P. Tie, nad ktorými x nevyhrá, presunieme z P do S.

Akonáhle sa fronta S vyprázdni, máme všetky programy rozdelené do V a P a môžeme skončiť.

Prvé kvadratické riešenie

V tomto okamihu je už ľahké napísať program s časovou zložitosťou $O(N^2)$.

Načítame vstup a do dvojrozmerného poľa si zaznačíme, kto nad kým môže vyhrať. (Na začiatku môže vyhrať každý nad každým, ako postupne čítame vstup, značíme si, kto nad kým vyhrať nemôže.)

Odsimulujeme jeden turnaj tak, že najskôr hrá 1 s 2, potom víťaz s 3, a tak ďalej až kým nezostane len jeden program.

Teraz opakujeme dokola cyklus z predchádzajúcej časti. Pridať nový prvok do x vieme v konštantnom čase a upraviť množinu P v čase $O(N)$. (Na to si stačí množinu P reprezentovať ako pole, kde máme o každom prvku zaznačené, či do P patrí alebo nie.)

Keďže v každom opakovaní cyklu pridáme do V jeden program, bude opakovaní najviac $N - 1$. A teda celková časová zložitosť bude $O(N^2)$.

Aby sme dosiahli lepšiu časovú zložitosť, potrebujeme zlepšiť dve miesta v našom programe: Prvou je vedieť aj bez pamäte $O(N^2)$ efektívne odsimulovať turnaj, druhou je šikovnejšie upraviť množinu P.

Prienik dvoch utriedených postupností

Majme dve postupnosti A a B celých čísel, pričom obe sú utriedené podľa veľkosti a žiadna neobsahuje to isté číslo dvakrát. Chceme nájsť ich prienik, teda tie čísla, ktoré sa vyskytujú v oboch postupnostiach.

Riešenie je ľahké: Začnime tým, že sa pozrieme na prvé členy postupností. Ak sú oba rovnaké, dáme príslušnú hodnotu na výstup a oba zahodíme. Ak je jeden z nich menší, môžeme ho rovno zahodiť – v druhej postupnosti taká hodnota určite nebude. Toto celé teraz opakujeme dokola až kým sa nám jedna z postupností neminie.

Každý krok vieme spraviť v konštantnom čase. (Postupnosti A a B v skutočnosti nebudeme meniť, len si budeme pamätať, kde v ktorej z nich je práve prvý nevyhodený prvok.) A keďže v každom kroku niečo vyhodíme, bude počet krokov nanajvýš rovný súčtu dĺžok postupností.

Reprezentácia množiny porazených

Ako sme už naznačili, množinu P si budeme pamätať ako utriedené pole čísel programov, ktoré do nej patria.

Na začiatku toto máme „zadarmo“, programy, nad ktorými v vyhral, máme presne v tejto podobe dané na vstupe.

Keď teraz pridávame do V nový program, spravíme vyššie uvedeným postupom prienik doterajšej množiny P a množiny programov, nad ktorými pridávaný program vyhrá.

Čo sme týmto získali? Ukážeme, ako dlho budú trvať dokopy všetky zmeny množiny P. Nech q_i je počet programov, o ktorých máme na vstupe povedané, že nad programom i vyhrajú. Zjavne $q_1 + \cdots + q_N = M$. Všimnime si teraz ľubovoľný program p, ktorý sme na začiatku umiestnili do množiny P. Tento program v nej vydrží najviac q_p krokov, a teda ho budeme spracúvať najviac $q_p + 1$ krát.

Dokopy všetky prvky z P budeme teda spracúvať nanajvýš $(q_1 + 1) + \cdots + (q_N + 1) = M + N$ krát. Preto celková časová zložitosť udržiavania množiny P bude $O(M + N)$.

Simulácia turnaja

Posledné, čo potrebujeme, je efektívne odsimulovať jeden turnaj, pričom si informácie o tom, kto s kým musí vyhrať, budeme pamätať len v podobe, v akej sú dané na vstupe.

Budeme mať jedno pole, v ktorom si pre každý program pamätáme, či ešte je v turnaji alebo už vypadol. Na začiatku nastavíme, že kandidátom k na výhru v turnaji je program 1. Z turnaja vyhodíme všetky programy, nad ktorými program 1 určite vyhrá.

Teraz budeme postupne prechádzať všetky programy. Zakaždým, keď narazíme na nejaký, ktorý ešte nevypadol, spravíme nasledovné veci:

V prvom rade, práve nájdený program p môže vyhrať nad doterajším kandidátom k. (Tie, ktoré nad k vyhrať nemôžu, sme už vyhodili.) Doterajšieho kandidáta z turnaja vyhodíme, odteraz bude kandidátom na výhru náš program p. A samozrejme, teraz z turnaja vyhodíme všetky programy, nad ktorými náš nový kandidát určite vyhrá. (Teda presnejšie, vyhodíme tie z nich, ktoré sme ešte nevyhodili.)

Keď takto prejdeme cez všetky programy, ostane nám už v turnaji len jeden program (aktuálny kandidát), a ten teda turnaj vyhral.

Zjavne sa na každý program pozrieme práve raz a každú informáciu zo vstupu použijeme najviac raz, preto je časová zložitosť simulácie $O(M + N)$.

Alternatívne vzorové riešenie

V čase $O(M + N)$ sa dá programy zoradiť do poradia, v ktorom platí, že každý program môže vyhrať nad programom, ktorý nasleduje bezprostredne po ňom.

Jedna možnosť ako to spraviť: Postupne pridávame programy do poradia. Nech je aktuálne poradie $p_1, \ldots, p_k$ a práve pridávame nový program p. Nájdeme najväčšie také j, že p_j môže vyhrať nad p (teda nepatrí medzi programy, nad ktorými p určite vyhrá). Umiestnime p do poradia tesne za p_j. Jediný špeciálny prípad: ak zistíme, že p porazí všetky p_i, dáme ho na začiatok poradia.

Zjavne máme opäť korektné poradie – p_j sme vybrali tak, aby vedelo vyhrať nad p, a navyše p určite vyhrá nad každým napravo od seba, teda aj s programom bezprostredne za ním (ak taký existuje).

Toto poradie vieme zostrojiť v čase $O(M + N)$, dokonca si na to vystačíme s dvoma obyčajnými poliami. V poli P na pozíciách 1 až k si budeme pamätať aktuálne poradie programov. Okrem toho budeme mať jedno pole B, v ktorom si o každom programe budeme pamätať, či nad ním aktuálny program p určite vyhrá alebo nie. Aby sme nemuseli pole B pri každom novom p celé nulovať, použijeme jeden jednoduchý trik.

Algoritmus bude fungovať nasledovne:

1. Vynuluj pole B, nastav k na 1 a $P[1]$ na 1.
2. Postupne pre $x = 2$ až N opakuj nasledujúce kroky:
3. Prejdi zoznam d_i programov, nad ktorými program x vyhrá.
   Do poľa B na príslušné pozície zapíš číslo x (toto je spomínaný trik).
4. Postupne od konca prechádzaj pole P, kým neprídeš na pozíciu j takú, že $B[P[j]] \neq x$ alebo na začiatok (kedy bude $j = 0$).
5. Prvky v poli P na pozíciách $j + 1$ až k posuň o 1 doprava.
   Do $P[j]$ ulož x a zväčši k o 1.

(Všimnite si, že všetky kroky 3 dokopy trvajú $O(d_1 + \cdots + d_n) = O(M)$, a že každý krok 4 trvá rádovo toľko isto ako jemu predchádzajúci krok 3.)

Pozrime sa teraz na naše poradie programov. Ak nejaký program môže vyhrať turnaj, môžu vyhrať turnaj aj všetky programy, ktoré sú v poradí pred

ním. Takže stačí nájsť hranicu takú, že naľavo od nej sú programy, ktoré turnaj vyhrať môžu, a napravo tie, ktoré vyhrať nemôžu.

Na hľadanie hranice opäť použijeme pozorovanie ♡, len sa nám tentokrát bude ľahšie implementovať. Keď teraz spracúvame nejaký program p, o ktorom vieme, že môže vyhrať turnaj, stačí nám nájsť v našom poradí prvý program q od konca, ktorý vie nad p vyhrať. Program q aj všetky naľavo od neho vedia turnaj vyhrať. Ak q leží napravo od doteraz nájdenej hranice, príslušne ju posunieme. No a hľadanie programu q je jednoduché – sú to opäť presne kroky 3 a 4 z vyššie uvedeného algoritmu.

Dokopy bude teda táto druhá fáza vyzerať nasledovne:

1. Nastav index k posledného nám známeho vyhrávajúceho programu na 1. Nastav index x vyhrávajúceho programu, ktorý treba spracovať, na 1.
2. Ak $x > k$, už sme spracovali všetky vyhrávajúce programy a nič nové sa nedozvieme, takže vieme, že turnaj môže vyhrať práve prvých k programov z nášho poradia.
3. Nájdi index j prvého programu od konca poradia, ktorý môže vyhrať nad programom $P[x]$. (Ak $P[x]$ nutne vyhrá nad každým napravo od seba, bude $j = x$.)
4. Ak $j > k$, nastav k na j. Zväčši x o 1 a pokračuj krokom 2.

Pomalšie riešenia

Na dvojrozmerné pole, v ktorom máme zaznačené, kto nad kým môže vyhrať, sa môžeme dívať ako na maticu susednosti orientovaného grafu.

Vo vzorovom riešení sme zdôvodnili, že ak v vie vyhrať turnaj, tak množinu všetkých víťazov dostaneme tak, že zoberieme:

1. v
2. každého kto vie vyhrať nad v
3. každého, kto vie vyhrať nad niekým z kroku 2
4. ...

V grafovej terminológii to vyjadríme ľahšie: možní víťazi turnaja sú práve tie programy, z ktorých je v našom grafe dosiahnuteľný vrchol v.

Druhé kvadratické riešenie teda dostávame nasledovne: Podobne ako v prvom riešení simuláciou turnaja nájdeme prvé v. Teraz otočíme všetky hrany a prehľadávaním (do hĺbky alebo do šírky) z v nájdeme všetky programy, ktoré môžu turnaj vyhrať.

V podobnom duchu môžeme dostať riešenie s časovou zložitosťou $O(N^3)$. Na takéto riešenie nepotrebujeme ani len pozorovanie ♡. Iba si stačí všimnúť (a dokázať), že vo vyššie zostrojenom grafe platí: Program p môže vyhrať turnaj práve vtedy, ak je z p dosiahnuteľný každý vrchol v grafe. Postupne pre každý vrchol teda spustíme prehľadávanie, ktorým nájdeme všetky z neho dosiahnuteľné vrcholy.

Listing programu:

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int N; // pocet programov
vector< vector<int> > vyhra; // pre kazdy program zoznam tych, nad ktorymi vyhra

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

int simuluj() {
  vector<bool> hra(N+1,true); // o kazdom programe, ci este hra v turnaji
  int kandidat = 1;
  for (unsigned i=0; i<vyhra[kandidat].size(); i++)
    hra[ vyhra[kandidat][i] ] = false;
  for (int dalsi=2; dalsi<=N; dalsi++)
    if (hra[dalsi]) {
      hra[ kandidat ] =0;
      kandidat = dalsi;
      for (unsigned i=0; i<vyhra[kandidat].size(); i++)
        hra[ vyhra[kandidat][i] ] = false;
    }
  return kandidat;
}

int main() {
```

```
  nacitaj();
  // najdeme prveho mozneho vitaza
  int v = simuluj();

  // inicializujeme V a P
  vector<int> V, P;
  V.push_back(v);
  P = vyhra[v];

  // do fronty S vlozime vsetky prvky, co nie su vo V ani P
  queue<int> S;
  vector<int> tmp(N+1);
  tmp[v] = 1;
  for (unsigned i=0; i<P.size(); i++) tmp[P[i]] = 1;
  for (int i=1; i<=N; i++) if (!tmp[i]) S.push(i);

  // kym mame nieco vo fronte S, pridame to do V a preratame P
  while (!S.empty()) {
    int x = S.front(); S.pop();
    V.push_back(x);
    // ideme preratat P + co vyhodime z P, ide do S
    vector<int> newP;
    unsigned a=0, b=0;
    while (a<P.size() && b<vyhra[x].size()) {
      if (P[a] == vyhra[x][b]) { newP.push_back(P[a]); a++; b++; continue; }
      if (P[a] < vyhra[x][b]) { S.push(P[a]); a++; continue; }
      if (P[a] > vyhra[x][b]) { b++; continue; }
    }
    P = newP;
  }
  for (unsigned i=0; i<V.size(); i++) cout << V[i] << endl;
}
```

A-II-4 Prekladacie stroje

Podúloha a)

Použijeme postup podobný tomu z domáceho kola.

V prvom kroku zostrojíme prekladací stroj, ktorý kopíruje vstup na výstup a navyše na ľubovoľné miesto v reťazci vie dopísať ľubovoľne veľa znakov c. Tento stroj môže vyzerať nasledovne:

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

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

Množina $M_2 = Z_1(M_1)$ teda obsahuje práve všetky reťazce z písmen a, b, c, kde je rovnako veľa a a b.

Analogicky teraz zostrojíme druhý stroj, ktorý vyrobí tie reťazce, kde bude rovnako veľa b a c, a ľubovoľne veľa a. Pre zmenu môžeme túto množinu reťazcov zostrojiť z práve zostrojenej M_2 tak, že cyklicky zameníme všetky a za b, b za c a c za a. Toto robí napríklad tento prekladací stroj:

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

Teraz sme teda zostrojili množinu $M_3 = Z_2(M_2)$.

No a je evidentné, že prienikom množín M_2 a M_3 dostaneme práve hľadanú množinu G.

Podúloha b)

Základným trikom pri riešení tejto úlohy je uvedomiť si, že ak má byť výstupná množina reťazcov dostatočne jednoduchá, vstup vlastne na nič nepotrebujeme – vieme ju vygenerovať aj „bez dopomoci“.

V našom prípade bude teda výsledný prekladací stroj pracovať nasledovne:

1. Prečíta celý vstup a zahodí ho, teda zatiaľ nič nevypíše.
   (Podľa definície stroja toto musíme urobiť, preklad je platný len ak spracujeme celý vstupný reťazec!)

2. Začneme generovať číslo, pričom si v stave pamätáme, aký zvyšok dáva doteraz zapísané číslo po delení siedmimi. Ukončovací stav bude zodpovedať zvyšku 0 – teda prestať generovať môžeme (a nemusíme) práve vtedy, keď sme vygenerovali číslo deliteľné siedmimi.

Treba si dať ešte pozor na to, aby naše číslo nezačínalo nulou, toto vieme ale elegantne ošetriť napríklad v okamihu, keď zistíme, že sme dočítali vstupný reťazec.

Výsledný prekladací stroj bude vyzerať takto:

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

Slovný popis: Pravidlami v prvom riadku vieme prečítať celé vstupné slovo a nič nezapísať.

Pravidlo v druhom riadku použijeme práve raz, a to po dočítaní vstupného slova. Na výstup zapíšeme prvú cifru čísla (všimnite si, že musí byť kladná) a nastavíme si stav na jej zvyšok po delení siedmimi.

Teraz už môžeme používať len pravidlá v treťom riadku. Každé z nich dopíše na koniec výsledného reťazca nejakú novú cifru y. Matematicky vieme operáciu „pridaj na koniec čísla cifru y“ povedať aj „vynásob číslo desiatimi a pripočítaj k nemu y“. Ak teda doteraz zapísané číslo malo zvyšok po delení siedmimi rovný x, nové číslo bude mať rovnaký zvyšok ako $10x+y$. A presne do zodpovedajúceho stavu prejde náš prekladací stroj.

Z argumentu v predchádzajúcom odseku vyplýva, že každé číslo, ktoré bude vo výslednej množine $Z(X)$, je naozaj deliteľné siedmimi.

No a keďže na výstup vieme vypísať ľubovoľné kladné celé číslo, určite vie náš stroj vyrobiť každé číslo deliteľné siedmimi, a po jeho vypísaní bude určite v ukončovacom stave 0. Preto naozaj $Y = Z(X)$.

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY TRETÍ ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX