Slovenská komisia Matematickej olympiády
FPEDaS ŽU, Veľký diel, 010 26 Žilina

MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

55. ročník, školský rok 2005/2006

Riešenia úloh 1. kola kategórie P

Tento pracovný materiál nie je určený priamo študentom — účastníkom olympiády. Má pomôcť učiteľom na školách pri príprave konzultácií a pracovných seminárov pre riešiteľov súťaže, členom krajských výborov MO slúži ako podklad pre opravovanie úloh domáceho kola MO kategórie P. **Študentom možno tieto komentáre poskytnúť až po termíne stanovenom pre odovzdanie riešení úloh domáceho kola MO kategórie P** ako informáciu, ako bolo treba úlohy správne riešiť a pre ich odbornú prípravu na účasť v krajskom kole súťaže.

P–I–1

Tento príklad má veľa rôzne rýchlych riešení, niektoré z nich si postupne ukážeme.

Priamočiare riešenie je zostrojiť si podľa vstupu *bipartitný graf* (vrcholy jednej partície sú hráčove pluky, druhú partíciu tvoria pluky počítača, hrana predstavuje priradenie plukov, pri ktorom hráčov pluk vyhrá) a v tomto grafe nájsť *najväčšie párovanie* – najväčšiu množinu hrán, v ktorej žiadne dve nemajú spoločný vrchol. Existujú známe algoritmy riešiace túto úlohu, najznámejší a najjednoduchší z nich je založený na hľadaní zlepšujúcich ciest. Tento algoritmus má časovú zložitosť $O(MN)$, v najhoršom prípade teda $O(N^3)$. Keďže ale existujú aj lepšie riešenia pre túto úlohu, nebudeme sa tu algoritmami na hľadanie párovania podrobnejšie zaoberať.

To, čo chceme dosiahnuť, je vybrať čo najviac navzájom disjunktných dvojíc (hráčov pluk, pluk počítača), v ktorých má hráčov pluk viac vojakov ako pluk počítača.

Utrieďme pluky hráča aj pluky počítača podľa počtu vojakov nerastúco (t.j. každého prvý pluk bude najväčší, atď.).

Pozorovanie 1. Existuje optimálne riešenie, v ktorom keď zoradíme hráčove pluky podľa veľkosti, budú aj im zodpovedajúce pluky počítača zoradené podľa veľkosti.

Prečo je tomu tak? Zoberme ľubovoľné optimálne riešenie R. Dokážeme si, že keď teraz prehádžeme pluky počítača tak, aby boli zoradené podľa veľkosti, opäť dostaneme prípustné riešenie R'.

Nech P je k-ty najväčší vybratý pluk počítača. V riešení R mu je priradený nejaký hráčov pluk, ktorý je od neho väčší. Máme ešte $K-1$ plukov počítača, ktoré sú väčšie ako P. Každý z nich má v R priradený nejaký hráčov pluk, všetky tieto hráčove pluky sú od nich väčšie, a teda sú väčšie aj od P. Vieme teda už o k hráčovych plukoch, ktoré sú od P väčšie. Preto aj v riešení R' (kde je pluku P priradený k-ty najväčší hráčov pluk) bude P zjavne od svojej „dvojice“ menší.

Toto pozorovanie nám stačí na napísanie riešenia pomocou dynamického programovania: Optimálne riešenie pre prvých x hráčovych plukov a prvých y plukov počítača vyzerá buď tak, že x-tý pluk hráča porazí y-ty pluk počítača (ak sa to dá) a zvyšné pluky priradíme (už spočítaným) optimálnym spôsobom, alebo nevyberieme x-tý pluk hráča, alebo nevyberieme y-ty pluk počítača.

Priamočiara implementácia tohto riešenia potrebuje čas aj pamäť $O(N^2)$. Pamäťové nároky sa pri veľkej snahe dajú zmenšiť na $O(N)$. Opäť, detaily nechávame na čitateľa, keďže ešte máme na sklade pár lepších riešení tejto úlohy.

Ukážeme si teraz ľahšie napísateľné riešenie, ktoré bude potrebovať čas $O(N^2)$ a pamäť $O(N)$.

Pozorovanie 2. Existuje optimálne riešenie, v ktorom vyberieme najväčších K hráčovych plukov a najmenších K plukov počítača.

Dôkaz. Zoberme ľubovoľné optimálne riešenie. Kým sa to dá, nahrádzajme vybratý pluk počítača za menší nevybratý. Zjavne stále dostávame rovnako dobré prípustné riešenia. Keď sa to už nedá, zjavne máme vybratých niekoľko najmenších plukov počítača. Teraz ešte analogicky „zväčšíme“ vybraté hráčove pluky a vyhrali sme.

Predchádzajúce dve pozorovania dokopy nám stačia na triviálne napísateľné kvadratické riešenie – jednoducho postupne pre každé K skontrolujeme, či dostaneme prípustné riešenie, ak K najväčším hráčovym plukom priradíme (v takom istom poradí) K najmenších plukov počítača.

Ešte stále existuje aj lepšie riešenie.

Rovnako ako v predchádzajúcom prípade si uvedomíme, že môžeme vybrať niekoľko najväčších hráčovych plukov. Avšak postupne (začínajúc najväčším hráčovym plukom) každému z nich priradíme čo najväčší pluk počítača, ktorý ešte dokáže poraziť.

Zostáva zodpovedať dve otázky. Prvá z nich je, prečo to funguje.

Zoberme ľubovoľné optimálne riešenie, v ktorom sme vybrali niekoľko najväčších hráčovych plukov a v ktorom sú pluky oboch hráčov zoradené podľa veľkosti. Postupne od najväčšieho sa pozerajme na pluky počítača a každý z nich skúšajme nahradiť **väčším** nepoužitým, kým sa to dá. Takto zjavne dostaneme rovnako dobré prípustné riešenie – a ľahko nahliadneme, že práve toto riešenie nájde aj vyššie popísaný algoritmus.

Druhá otázka je, ako to efektívne implementovať. Na to si stačí uvedomiť, že každému ďalšiemu hráčovmu pluku priradíme menší pluk počítača ako predchádzajúcemu. Preto si stačí udržiavať index posledného priradeného pluku počítača.

Takto nájdeme optimálne priradenie v lineárnom čase. Nesmieme ale zabudnúť, že sme museli pluky utriediť podľa veľkosti. Preto celková časová zložitosť tohto riešenia je $O(N \log N)$.

```
#include <stdio.h>
#include <stdlib.h>

int N; // pocet plukov
int A[11000], B[11000]; // pocty vojakov

// porovnavacia funkcia pre inty
int icmp(int *a, int *b) { return (*b) - (*a); }

int main(void) {
  int i,H,C;

  // nacitame vstup
  scanf("%d ",&N);
  for (i=0;i<N;i++) scanf("%d ",&A[i]);
  for (i=0;i<N;i++) scanf("%d ",&B[i]);

  // utriedime vstup
```

```
  qsort(A,N,sizeof(int),icmp);
  qsort(B,N,sizeof(int),icmp);

  // najdeme optimalne priradenie
  H = 0; C = 0;
  while (1) {
    // najdeme prvy pluk pocitaca, ktory porazime
    while (A[H]<=B[C] && C<N) C++;
    // ak taky nie je, koncime
    if (C==N) break;
    // ideme najst par dalsiemu hracovmu pluku
    H++; C++;
  }

  // vypiseme, kolkym hracovym plukom sme nasli dvojicu
  printf("%d\n",H);
  return 0;
}
```

P–I–2

Situáciu zo zadania si predstavíme ako ohodnotený orientovaný graf. Vrcholmi budú lokality, hranami teleporty medzi lokalitami. Každá hrana je ohodnotená číslom, ktoré reprezentuje posun v čase pri prechode danou hranou (budeme ho volať dĺžka). Úlohou je nájsť sled[1] z vrchola 1 do vrchola N s najmenším súčtom ohodnotení hrán (budeme ho volať najkratší), prípadne vypísať, že takýto sled neexistuje.

Najskôr si všimnime, že ak medzi dvoma vrcholmi vedie viac hrán (tým istým smerom), tak stačí uvažovať tú s najmenšou dĺžkou (inak by sme vedeli sled skrátiť výmenou hrany za kratšiu). Ďalej si všimnime, že náš graf môže obsahovať aj hrany so zápornou dĺžkou. Môžu teda nastať dve situácie, kedy hľadaný najkratší sled neexistuje. Buď sa z 1 do N po hranách grafu nevieme dostať vôbec, alebo existuje sled z vrchola 1 do N taký, že obsahuje cyklus so záporným súčtom dĺžok. (Po takomto cykle môžeme chodiť dookola a vždy zmenšovať celkovú dĺžku sledu. Preto neexistuje najkratší sled – od každého sledu totiž vieme vyrobiť kratší.)

Riešenie 1, Floyd-Warshallov algoritmus. Graf si zapíšeme do dvojrozmerného poľa `G`, pričom `G[i][j]` bude dĺžka hrany z vrchola i do vrchola j (alebo ∞, ak taká hrana neexistuje). Algoritmus potom vyzerá takto:

```
for k:=1 to N do
  for i:=1 to N do
    for j:=1 to N do
      if G[i][j] > G[i][k]+G[k][j] then
        G[i][j] := G[i][k]+G[k][j];
```

Po dobehnutí algoritmu je v `G[i][j]` dĺžka najkratšieho sledu z i do j (prípadne ∞, ak žiaden neexistuje). Navyše, ak `G[i][i]` je záporné pre nejaké i, tak vrchol i leží na nejakom

[1] Sled je postupnosť vrcholov $v_1, \ldots, v_k$, taká že medzi v_i a v_{i+1} je hrana.

zápornom cykle. Ak tento vrchol leží na nejakom slede z 1 do N (teda `G[1][i]` a `G[i][N]` nie sú ∞), tak existuje ľubovoľne krátky sled z 1 do N.

Idea algoritmu je v dynamickom programovaní. Ak vonkajší cyklus prebehol k krát, tak `G[i][j]` je dĺžka najkratšieho sledu z i do j takého, že ako vnútorné vrcholy používa len vrcholy z množiny $\{1, \ldots, k\}$.

Časová zložitosť tohto algoritmu je $O(N^3)$, pamäťová $O(N^2)$.

Riešenie 2, Bellman-Fordov algoritmus. Hrany v tomto algoritme si pamatáme jednoducho, napr. v 3 poliach, kde `a[i]` je začiatok, `b[i]` je koniec a `t[i]` je dĺžka i-tej hrany. Idea tohto algoritmu spočíva tiež v dynamickom programovaní. Nech `D[l][i]` je dĺžka takého najkratšieho sledu z 1 do i, ktorý používa práve l hrán. Zrejme `D[0][i]` je ∞ pre všetky i okrem $i = 1$, pre ktoré je to 0. Nech teraz poznáme `D[l-1][i]` pre všetky i a nejaké $l > 0$. Je ľahko vidieť, ako teraz vypočítame `D[l][i]` pre ľubovoľné i. V najkratšom slede používajúcom l hrán je nejaká hrana posledná a zvyšok je najkratší sled používajúci $l - 1$ hrán, končiaci v začiatku l-tej hrany. Jednoducho vyskúšame všetky možnosti pre túto poslednú hranu. Máme teda:

$$\texttt{D[l][i]} = \min_{1 \le k \le M} \left\{ \texttt{D[l-1][a[k]]} + \texttt{t[k]},\ \text{kde } \texttt{b[k]} = i \right\}$$

Najjednoduchšie sa to (pri našej reprezentácii grafu) počíta tak, že prejdeme postupne cez všetky hrany a počítame jednotlivé minimá pre všetky vrcholy súčasne.

Vieme, že ak existuje najkratší sled z 1 do N, tak potom bude mať nanavýš $N - 1$ hrán (lebo neobsahuje cyklus). Výsledkom je teda minimum z `D[l][N]` pre $l = 1, \ldots, N - 1$. Ak je táto hodnota ∞, tak žiaden sled neexistuje. Ešte potrebujeme overiť, či sa nemôžeme dostať do záporného cyklu. Takýto cyklus môže obsahovať najviac N hrán (predstavme si graf s hranami (1,2,1), (2,3,1), …, ($N-1$,N,1), (N,1,$-N$)). Ak teda existuje sled obsahujúci záporný cyklus, tak existuje aj sled dĺžky nanajvýš $2N - 1$, ktorý tento cyklus obsahuje. Úpravu vzdialeností teda spustíme ešte N krát. Ak minimum z `D[l][N]` pre $N \le l \le 2N-1$ je menšie ako výsledok, ktorý sme našli predtým, potom naozaj existuje sled z 1 do N, ktorý obsahuje záporný cyklus.

Takto určite odhalíme všetky vrcholy, ktoré ležia na nejakom (z vrcholu 1 dosiahnuteľnom) zápornom cykle. Navyše takto odhalíme aj niektoré z vrcholov, ktoré sú z nejakého takéhoto záporného cyklu dosiahnuteľné – ale nie nutne všetky! (Ako príklad uvažujme graf s zápornou hranou z 1 do 1 a s oveľa oveľa drahšou kladnou hranou z 1 do 2.)

Aby sme teda zistili, či existuje ľubovoľne krátky sled z 1 do N, potrebujeme zistiť, či je vrchol N dosiahnuteľný z niektorého z (dosiahnuteľných) záporných cyklov, teda či je dosiahnuteľný z niektorého z vrcholov, ktoré sme našli vyššie popísaným postupom. Toto už vieme spraviť jedným jednoduchým prehľadávaním.

Na záver ešte jedno zjednodušenie. Uvedomme si, že nás nezaujímajú presne dĺžky sledov s práve k hranami. Preto nám stačí jednorozmerné pole `D[i]` a jednotlivé úpravy stačí robiť len na ňom. Premyslite si, že výsledok to nezmení (aj keď jednotlivé iterácie budú vyzerať inak).

Časová zložitosť tohto algoritmu je $O(NM)$, pamäťová $O(N + M)$.

Poznámka pre skusených: Ak poznáte Dijkstrov algoritmus, tak isto viete, že je rýchlejší od oboch uvedených algoritmov, avšak nefunguje na grafoch so zápornými hranami! Skúste si vymyslieť kontrapríklad.

Poznámka pre ešte skusenejších: Ak by sme hľadali najkratšiu cestu (teda sled, v ktorom vrcholy sa nemôžu opakovať), tak daný problém by sa už stal NP-ťažkým problémom (všetky známe polynomiálne algoritmy na hľadanie najkratšej cesty totiž fungujú len na špeciálnych typoch grafov, väčšinou ide o grafy bez záporných cyklov).

program teleport;

```
const INF=1000000000;
      MAXN=1000;
      MAXM=50000;
var
    { rozmery grafu }
    N,M:integer;
    { hrany }
    a:array[1..MAXM] of integer;
    b:array[1..MAXM] of integer;
    t:array[1..MAXM] of integer;
    { vzdialenosti }
    D:array[1..MAXN] of longint;
    D2:array[1..MAXN] of longint;
    { pomocne premenne }
    i,l,k:integer;
    vysledok:longint;
    { fronta na prehladavanie do sirky }
    Q:array[1..MAXN] of longint;
    qs,qf:longint;

procedure nacitaj;
var f:text;
begin
  assign(f,'teleport.in');
  reset(f);
  readln(f,N,M);
  for i:=1 to M do
    readln(f,a[i],b[i],t[i]);
  close(f);
end;

procedure pocitaj;
var f:text;
begin
  assign(f,'teleport.out');
  rewrite(f);
  D[1]:=0;
  for i:=2 to N do
    D[i]:=INF;

  for l:=1 to N-1 do
    for k:=1 to M do
      if D[b[k]]>D[a[k]]+t[k] then
        D[b[k]]:=D[a[k]]+t[k];
  vysledok:=D[N];

  if vysledok=INF then
    writeln(f,'Vedci umru od hladu')
  else begin
```

```
    for i:=1 to D do D2[i] := D[i];

    for l:=1 to N do
      for k:=1 to M do
        if D2[b[k]]>D2[a[k]]+t[k] then
          D2[b[k]]:=D2[a[k]]+t[k];

    qs:=1; qf:=1;
    for i:=1 to N do if D2[i] < D[i] then begin
      Q[qf]:=i; inc(qf); D2[i]:=1;
    end else D2[i]:=0;

    { z lenivostnych dovodov aj prehladavanie do sirky
     spravime v inak otrasnom case O(MN) }
    while qs<qf do begin
      k:=Q[qs]; inc(qs);
      for i:=1 to M do if a[i]=k and not D2[b[i]] then begin
        D2[b[i]]:=1;
        Q[qf]:=b[i];
        inc(qf);
      end;
    end;

    if D2[N] then
      writeln(f,'Vedci spoznaju zaciatok vesmiru')
    else
      writeln(f,vysledok);
  end;

  close(f);
end;

begin
  nacitaj;
  pocitaj;
end.
```

P–I–3

Na riešenie tejto úlohy použijeme postup známy pod názvom *dynamické programovanie* – budeme riešiť úlohu zo zadania (a jej drobne zmenené verzie) postupne pre rôzne časti cestnej siete, pričom z výsledkov pre menšie časti budeme počítať výsledky pre väčšie časti mapy. Ako to celé bude fungovať sa dočítate v ďalšom texte.

Cestnú sieť v kráľovstve môžeme tradičným spôsobom reprezentovať grafom – vrcholy grafu budú mestá, hrany cesty medzi nimi. Zo zadania vieme, že tento graf je strom, teda má N vrcholov, práve $N - 1$ hrán a je súvislý.

Ľubovoľný z vrcholov stromu nazveme jeho koreňom. Všetkých jeho susedov nazveme jeho synmi, ich ostatní susedia budú zase ich synmi, atď. (Môžeme si to celé predstaviť tak, že celý strom zavesíme za koreň. Otcom vrcholu je ten jeho sused, ktorý je nad ním, ostatní susedia budú jeho synmi.)

Od tohto okamihu budeme pod slovom *podstrom s koreňom v* rozumieť tú časť nášho stromu, z ktorej sa do koreňa vieme dostať len cez vrchol v. Keď budeme hovoriť o *podstrome*, myslíme tým podstrom s ľubovoľným koreňom.[2]

Názov *skupina posádok* bude označovať množinu vrcholov s posádkami, ktorá je súvislá a nesusedí s žiadnym ďalším vrcholom s posádkou.

Všimnime si, ako vyzerá optimálne riešenie. Pre koreň máme dve možnosti: buď tam posádka je, alebo tam nie je. Ak tam nie je, ostalo nám niekoľko *samostatných* podstromov, v každom z nich sú posádky rozmiestnené optimálnym spôsobom. Čo ak tam posádka je? Všimnime si skupinu posádok obsahujúcu koreň. V žiadnom z vrcholov, ktoré s ňou susedia, posádka byť nemôže. Keď odstránime všetky tieto vrcholy z grafu, opäť nám ostane niekoľko *samostatných* podstromov. A opäť v každom z týchto podstromov sú posádky rozmiestnené optimálne.

Keby sme teda vedeli optimálne spôsoby rozmiestnenia posádok pre všetky podstromy, vedeli by sme teraz vyskúšaním konečného počtu možností nájsť optimálny spôsob rozmiestnenia posádok pre celý strom.

Lenže aj každý z podstromov je strom a môžme preň zopakovať celú túto úvahu.

No a princíp *dynamického programovania* je o tom, že sa na problém pozrieme z opačnej strany. Pre *listy* (vrcholy, ktoré nemajú žiadneho syna) je riešenie triviálne. No a postupne budeme spracúvať čím ďalej, tým väčšie podstromy, až kým nenájdeme optimálne riešenie pre celý strom.

Uvedomte si, že v okamihu, keď spracúvame nejaký podstrom, boli už všetky *jeho* podstromy spracované, a teda pre ne vieme optimálne riešenie.

Aby sa nám riešenie ľahšie implementovalo, urobíme nasledujúci trik:

Nech $A_{v,i}$ je hodnota najlepšieho riešenia pre podstrom s koreňom v, ak vieme, že skupina posádok, obsahujúca v, má veľkosť i. (Pritom i je od 0 do 3, $i = 0$ znamená, že vo v nie je posádka.)

Ďalej nech $A_v = \max_i A_{v,i}$ je hodnota najlepšieho riešenia pre podstrom s koreňom v.

Pripomeňme si zo zadania, že číslo b_v hovorí, koľko pridáva posádka vo v k bezpečnosti kráľovstva.

Ukážeme, ako ľahko spočítať tieto hodnoty pre v, ak ich poznáme pre všetkých jeho synov. Nech v má synov $s_1, \ldots, s_K$.

Zjavne $A_{v,0} = \sum_{i=1}^{K} A_{s_i}$ – každý z podstromov vyriešime optimálne.

Ďalej $A_{v,1} = b_v + \sum_{i=1}^{K} A_{s_i,0}$ – zarátame prínos vrcholu v, každý podstrom vyriešime optimálne s tým, že jeho vrchol musí zostať prázdny.

Podobne spočítame $A_{v,2}$ (len v jednom podstrome vyberieme $A_{s_i,1}$) a $A_{v,3}$ (v dvoch podstromoch vyberieme $A_{s_i,1}$ alebo v jednom $A_{s_i,2}$).

Čas potrebný na spracovanie jedného vrcholu je úmerný počtu jeho susedov, celkový čas je teda úmerný počtu vrcholov a hrán dokopy. Hrán je však len $N - 1$, preto je celková časová zložitosť $O(N)$.

```
#include <cstdio>
#include <vector> // pole dynamickej velkosti, size() vracia pocet prvkov
```

[2] Často sa za podstrom považuje ľubovoľný podgraf, ktorý je strom. Úplne korektne by sme mali naše podstromy volať napr. *podstrom indukovaný vrcholom v*. Čitateľ určite pochopí, že kvôli ľahšiemu vyjadrovaniu sme radšej zvolili takúto dohodu.

```
#include <queue>  // fronta
using namespace std;

int N; // pocet vrcholov

// cislo otca a zoznam synov pre kazdy z vrcholov
vector<int> otec;
vector< vector<int> > syn;

vector<int> b; // prispevky vrcholov k bezpecnosti

vector<int> A0, A1, A2, A3; // optimalne riesenia, vid pokec

// pomocne info pre vypis riesenia
vector< vector<int> > kto2, kto3;
int space=0;

void nacitaj(void) {
  // docasna premenna na ulozenie grafu, kym neurcime koren
  vector< vector<int> > graf;

  scanf("%d ",&N);
  graf.resize(N);

  // nacitame hrany
  int x,y;
  for (int i=0; i<N-1; i++) {
    scanf("%d %d ",&x,&y);
    // vrcholy budeme cislovat od 0
    x--; y--;
    // graf[x] je pole so susedmi vrcholu x, na koniec pola pridame y (a naopak)
    graf[x].push_back(y); graf[y].push_back(x);
  }

  // nacitame vahy vrcholov
  b.resize(N);
  for (int i=0; i<N; i++) scanf("%d ",&b[i]);

  // vrchol 0 bude koren, spustime z neho prehladavanie do sirky
  otec.resize(N,-1);
  otec[0]=0;

  queue<int> Q;
  Q.push(0);

  while (!Q.empty()) {
    int kde = Q.front(); Q.pop();
    for (int i=0; i<graf[kde].size(); i++)
      if (otec[ graf[kde][i] ] == -1) {
```

```
        otec[ graf[kde][i] ] = kde;
        Q.push( graf[kde][i] );
      }
  }

  // otcov mame v poriadku, doplnime synov
  syn.resize(N);
  for (int i=0; i<N; i++)
    for (int j=0; j<graf[i].size(); j++)
      if (graf[i][j] != otec[i])
        syn[i].push_back( graf[i][j] );

  // vynulujeme pamatane optimalne riesenia
  A0.resize(N,0);
  A1.resize(N,0);
  A2.resize(N,0);
  A3.resize(N,0);
  kto2.resize(N);
  kto3.resize(N);
}

void pis_cislo(int ake) {
  if (space) printf(" ");
  space=1;
  printf("%d",ake);
}

// vypis optimalneho riesenia pre podstrom s korenom "v",
// pricom skupina posadok obsahujuca koren ma velkost "ako"
void vypis(int v, int ako) {
  if (ako>0) pis_cislo(v+1);
  switch (ako) {
    case 0:
      // pre kazdeho syna vyberieme a vypiseme najlepsiu moznost
      for (int i=0; i<syn[v].size(); i++) {
        int toto=max(
          max( A0[ syn[v][i] ] , A1[ syn[v][i] ] ) ,
          max( A2[ syn[v][i] ] , A3[ syn[v][i] ] ) );
        if (A0[ syn[v][i] ]==toto) { vypis(syn[v][i],0); continue; }
        if (A1[ syn[v][i] ]==toto) { vypis(syn[v][i],1); continue; }
        if (A2[ syn[v][i] ]==toto) { vypis(syn[v][i],2); continue; }
        if (A3[ syn[v][i] ]==toto) { vypis(syn[v][i],3); continue; }
      }
    break;
    case 1:
      // pre kazdeho syna moznost s prazdnym korenom
      for (int i=0; i<syn[v].size(); i++) vypis( syn[v][i] , 0 );
    break;
    case 2:
```

```
      if (A2[v]==0) return;
      // pre jedneho syna moznost s 1 posadkou, pre ostatne 0
      vypis( kto2[v][0] , 1 );
      for (int i=0; i<syn[v].size(); i++)
        if ( syn[v][i] != kto2[v][0] )
          vypis( syn[v][i], 0 );
    break;
    case 3:
      if (A3[v]==0) return;
      // pre niekolko synov specialne moznosti
      if (kto3[v].size()==1) {
        vypis( kto3[v][0] , 2 );
      } else {
        vypis( kto3[v][0] , 1 );
        vypis( kto3[v][1] , 1 );
      }
      // zvysok podstromov ma prazdny koren
      for (int i=0; i<syn[v].size(); i++) {
        int ok=1;
        for (int j=0; j<kto3[v].size(); j++)
          if (kto3[v][j] == syn[v][i])
            ok=0;
        if (!ok) continue;
        vypis( syn[v][i], 0 );
      }
    break;
  }
}

int main(void) {
  nacitaj();

  // pre kazdy vrchol si pamatame, kolko jeho synov este treba spracovat
  vector<int> treba(N);
  for (int i=0; i<N; i++) treba[i] = syn[i].size();

  // vrcholy, ktore uz mozeme spracovat, cakaju vo fronte
  queue<int> Q;
  for (int i=0; i<N; i++) if (!treba[i]) Q.push(i);

  // kym nie sme hotovi, spracuvame vrcholy
  while (!Q.empty()) {
    int kde = Q.front(); Q.pop();

    // ak uz budeme moct spracovat otca, pridame do fronty
    treba[ otec[kde] ]--;
    if (!treba[ otec[kde] ]) Q.push( otec[kde] );

    A0[kde]=0;
```

```
for (int i=0; i<syn[kde].size(); i++) {
  int toto=max(
        max( A0[ syn[kde][i] ] , A1[ syn[kde][i] ] ) ,
        max( A2[ syn[kde][i] ] , A3[ syn[kde][i] ] ) );
  A0[kde] += toto;
}

A1[kde]=b[kde];
for (int i=0; i<syn[kde].size(); i++) A1[kde] += A0[ syn[kde][i] ];

if (syn[kde].size() == 0) continue;

A2[kde]=A1[kde];
int z1 = 0;
// najdeme ten podstrom, kde pridanim posadky najviac ziskame
for (int i=1; i<syn[kde].size(); i++)
  if (   A1[ syn[kde][i] ] - A0[ syn[kde][i] ]
       > A1[ syn[kde][z1] ] - A0[ syn[kde][z1] ])
    z1 = i;
A2[kde] += A1[ syn[kde][z1] ] - A0[ syn[kde][z1] ];
kto2[kde].push_back( syn[kde][z1] );

A3[kde]=A1[kde];
int z2 = 0;
// najdeme ten podstrom, kde pridanim 2 posadok najviac ziskame
for (int i=1; i<syn[kde].size(); i++)
  if (   A2[ syn[kde][i] ] - A0[ syn[kde][i] ]
       > A2[ syn[kde][z2] ] - A0[ syn[kde][z2] ])
    z2 = i;
A3[kde] += A2[ syn[kde][z2] ] - A0[ syn[kde][z2] ];
kto3[kde].push_back( syn[kde][z2] );

if (syn[kde].size() == 1) continue;

// este najdeme dva podstromy, kde pridanim po 1 posadke najviac ziskame
// jeden z nich uz vlastne mame najdeny v z1, staci najst druhy najlepsi
int z3 = 0; if (z3==z1) z3++;
for (int i=0; i<syn[kde].size(); i++) {
  if (i==z1) continue;
  if (   A1[ syn[kde][i] ] - A0[ syn[kde][i] ]
       > A1[ syn[kde][z3] ] - A0[ syn[kde][z3] ])
    z3 = i;
}
int nove = A1[kde]
          + A1[ syn[kde][z1] ] - A0[ syn[kde][z1] ]
          + A1[ syn[kde][z3] ] - A0[ syn[kde][z3] ];
if (nove > A3[kde]) {
  A3[kde] = nove;
  kto3[kde].clear();
```

```
      kto3[kde].push_back( syn[kde][z1] );
      kto3[kde].push_back( syn[kde][z3] );
    }
  }

  // uz vieme hodnotu optimalneho riesenia
  int best=max( max( A0[0] , A1[0] ) , max( A2[0] , A3[0] ) );
  printf("%d\n",best);

  // rekurzivne vypiseme jedno riesenie
  if (A0[0]==best) { vypis(0,0); printf("\n"); return 0; }
  if (A1[0]==best) { vypis(0,1); printf("\n"); return 0; }
  if (A2[0]==best) { vypis(0,2); printf("\n"); return 0; }
  if (A3[0]==best) { vypis(0,3); printf("\n"); return 0; }

  return 0;
}
```

P–I–4

Časť a)

Nech dĺžka reťazca *ihla* je N a dĺžka reťazca *seno* je M. Ukážeme riešenie v čase $O(\log N + \log M)$.

Základná myšlienka: Keby nám niekto povedal, kde začína niektorý výskyt reťazca *ihla* v reťazci *seno*, pomocou volania **Both** ľahko overíme v čase $O(\log N)$, či hovorí pravdu: Podobne ako sme v príklade v zadaniach vedeli paralelne overiť všetky delitele čísla, môžeme paralelne porovnať N dvojíc písmen.

Keďže nám ale nik nepovie, kde začína nejaký výskyt reťazca *ihla*, pomocou **Some** paralelne overíme všetky možné začiatky a úspešne skončíme, ak niektorý z nich vyhovuje.

```
{ VSTUP:  ihla, seno : string; }

{ forall() paralelne spusti N kopii, v ktorych cislo=0..(N-1),
  uspesne skonci, ak vsetky uspesne skoncia }
procedure forall(var cislo : integer, N : integer);
var moc2, cifier, i, x : integer
begin
   { zistime, kolko ma N-1 cifier v dvojkovej sustave }
   moc2 := 1;
   cifier := 0;
   while (moc2 <= N-1) do begin moc2 := moc2 * 2; inc( cifier ); end;
   { vygenerujeme cisla od 0 do 2^cifier - 1 }
   cislo := 0;
   for i:=1 to cifier do begin Both(x); cislo := 2*cislo + x; end;
   if (cislo >= N) then Accept;
end;
```

```
{ exists() paralelne spusti N kopii, v ktorych cislo=0..(N-1),
  uspesne skonci, ak niektora z nich uspesne skonci }
procedure exists(var cislo : integer, N : integer);
var moc2, cifier, i, x : integer
begin
   { zistime, kolko ma N-1 cifier v dvojkovej sustave }
   moc2 := 1;
   cifier := 0;
   while (moc2 <= N-1) do begin moc2 := moc2 * 2; inc( cifier ); end;
   { vygenerujeme cisla od 0 do 2^cifier - 1 }
   cislo := 0;
   for i:=1 to cifier do begin Some(x); cislo := 2*cislo + x; end;
   if (cislo >= N) then Reject;
end;

var N, M : integer;
    zaciatok, pozicia : integer;

begin
   { zistime dlzky retazcov }
   N := length( ihla );
   M := length( seno );
   if (M < N) then Reject;

   { skusime, ci existuje mozny zaciatok }
   exists( zaciatok, M-N+1 );
   { paralelne sa pozrieme na vsetky pozicie retazca ihla }
   forall( pozicia, N );
   { overime, ci su zodpovedajuce si pismena rovnake }
   if (ihla[pozicia+1] = seno[zaciatok+pozicia+1]) then Accept;
   Reject;
end.
```

Časť b)

Priamočiare riešenie je vypočítať postupne všetky políčka pyramídy a hodnotu horného porovnať s V. Políčok je $N(N-1)/2$, preto takéto riešenie beží v čase $O(N^2)$.

Ešte stále sme ale nevyužili dve skutočnosti: To, že poznáme V – a samozrejme silu paralelizátora. Ukážeme si teraz riešenie, ktoré pobeží v čase $O(N)$.

Ak by sme poznali obe čísla v druhom riadku zhora, vieme ľahko overiť, či je V správny súčet celej pyramídy – jednoducho ich sčítame. My ich však nepoznáme – iba vieme, že sú medzi 0 a 9 999 vrátane. Pomocou **Some** vyskúšame všetky dvojice takýchto čísel, ktoré (modulo 10 000) dávajú súčet V – t.j. skúsime „uhádnuť“ tie správne čísla, ktoré tam majú byť. Pre každú zo skúšaných možností potrebujeme o oboch nových číslach overiť, či patria na to miesto, kam sme ich práve umiestnili. Pomocou jedného volania **Both** overíme obe čísla naraz. No a už si

stačí len uvedomiť, že sme dostali opäť presne pôvodnú úlohu, len s pyramídou, ktorá má o 1 kratšiu základňu.

Keď sa takto dostaneme až k spodnému riadku, namiesto hádania sa už len jednoducho pozrieme na čísla zo vstupu.

Prečo naše riešenie funguje? Uvedomme si, že vlastne ideme v pyramíde „zhora dole“ a skúšame ju všetkými spôsobmi dopĺňať. Úspešne skončíme len vtedy, ak sa nám podarilo doplniť celú pyramídu. No a keďže pyramída je svojim spodným riadkom jednoznačne určená, toto sa dá len vtedy, ak je naozaj V na jej vrchu.

„Spracovať“ jednu úroveň pyramídy vieme v konštantnom čase, úrovní je $O(N)$, preto aj časová zložitosť nášho programu je $O(N)$.

```
{ VSTUP:  N, V : integer;  A : array[1..N] of integer; }

{ exists() paralelne spusti N kopii, v ktorych cislo=0..(N-1),
  uspesne skonci, ak niektora z nich uspesne skonci }
procedure exists(var cislo : integer, N : integer);
var moc2, cifier, i, x : integer
begin
   { zistime, kolko ma N-1 cifier v dvojkovej sustave }
   moc2 := 1;
   cifier := 0;
   while (moc2 <= N-1) do begin moc2 := moc2 * 2; inc( cifier ); end;
   { vygenerujeme cisla od 0 do 2^cifier - 1 }
   cislo := 0;
   for i:=1 to cifier do begin Some(x); cislo := 2*cislo + x; end;
   if (cislo >= N) then Reject;
end;

{ Over() overi, ci na policku [„riadok",„stlpec"] je cislo „hodnota"
  riadky cislujeme zdola, stlpce zlava, oboje od 1 }
procedure Over(riadok, stlpec, hodnota : integer);
var lave, prave, x : integer;
begin
   { ak uz sme dole, overime lahko }
   if (riadok = 1) then
      if (hodnota = A[stlpec]) then Accept else Reject;

   { skusime vsetky moznosti pre lave pod nim }
   exists(lave, 10000);
   { dopocitame prave pod nim }
   prave := hodnota - lave;
   if (prave < 0) then Inc( prave, 10000 );

   { overime, ci sme obe cisla vybrali spravne }
   Both(x);
   if (x=0) then
      Over( riadok-1, stlpec, lave )
   else
      Over( riadok-1, stlpec+1, prave );
```

```
end.

begin
   Over( N-1, 1, V );
end.
```

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

55. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia 1. kola kategórie P

Vydala IUVENTA
pre vnútornú potrebu Ministerstva školstva SR
Sadzba programom LaTeX
Zodpovedný redaktor M. Forišek