Tento pracovní materiál není určen přímo studentům – řešitelům olympiády. Má pomoci učitelům na školách při přípravě konzultací a pracovních seminářů pro řešitele soutěže, pracovníkům krajských výborů MO slouží jako podklad pro opravování úloh domácího kola MO kategorie P. Studentům poskytněte tato vzorová řešení až po termínu stanoveném pro odevzdání řešení úloh domácího kola MO-P jako informaci, jak bylo možné úlohy správně řešit, a pro jejich odbornou přípravu na účast v krajském kole soutěže.

P-I-1 Pluky

Tato úloha má mnoho různě rychlých řešení, některá z nich si postupně ukážeme.

Přímočarým řešením je sestrojit si na základě údajů ze vstupu *bipartitní graf* (vrcholy jedné jeho části představují hráčovy pluky, druhou část tvoří pluky počítače, hrana odpovídá takovému přiřazení pluků, při kterém hráčův pluk vyhraje) a v tomto grafu nalézt *maximální párování* – největší množinu hran, v níž žádné dvě nemají společný vrchol. Existují známé algoritmy řešící tento problém, nejznámější a nejjednodušší z nich je založen na hledání zlepšujících cest. Tento algoritmus má časovou složitost $O(MN)$, v nejhorším případě tedy $O(N^3)$. Protože ale pro tuto úlohu existují lepší řešení, nebudeme se zde algoritmy na hledání párovaní podrobněji zabývat.

Chceme vybrat co nejvíce navzájem disjunktních dvojic (hráčův pluk, pluk počítače), v nichž má hráčův pluk více vojáků než pluk počítače. Zkusíme seřadit pluky hráče i pluky počítače podle počtu vojáků do nerostoucího pořadí (tzn. první pluk každého bude největší, atd.).

Pozorování 1. Existuje optimální řešení, ve kterém platí: když seřadíme hráčovy pluky podle velikosti, budou také pluky počítače seřazeny podle velikosti.

Proč tomu tak je? Uvažujme libovolné optimální řešení R. Dokážeme, že pokud změníme pořadí pluků počítače tak, aby byly seřazeny podle velikosti, opět dostaneme přípustné řešení R'. Nechť P je k-tý největší vybraný pluk počítače. V řešení R mu je přiřazen nějaký hráčův pluk, který je větší. Máme ještě $k-1$ pluků počítače, které jsou větší než P. Každý z nich má v R přiřazen nějaký hráčův pluk, všechny tyto hráčovy pluky jsou větší než odpovídající pluky počítače, a proto jsou i větší než P. Víme tedy už o k hráčových plucích, které jsou všechny větší než P. Proto také v řešení R' (kde je pluku P přiřazen k-tý největší hráčův pluk) bude P zjevně menší než ten pluk hráče, který bude pluku P přiřazen.

Toto pozorování nám postačí k návrhu řešení úlohy pomocí dynamického programování: Optimální řešení pro prvních x pluků hráče a prvních y pluků počítače vypadá buď tak, že x-tý pluk hráče porazí y-tý pluk počítače (pokud je to možné) a zbývající pluky přiřadíme (už spočítaným) optimálním způsobem, nebo nevybereme x-tý pluk hráče, nebo nevybereme y-tý pluk počítače. Přímočará implementace tohoto řešení potřebuje čas i paměť $O(N^2)$. Paměťové nároky se při vynaložení většího úsilí dají snížit na $O(N)$. Detaily opět přenecháme čitateli, neboť ještě máme na skladě několik lepších řešení této úlohy. Ukážeme si nyní snadnější řešení, které bude potřebovat čas $O(N^2)$ a paměť $O(N)$.

Pozorování 2. Existuje optimální řešení, ve kterém vybereme K největších pluků hráče a K nejmenších pluků počítače.

Důkaz. Vezmeme libovolné optimální řešení. Dokud to půjde, budeme nahrazovat vybraný pluk počítače za menší nevybraný. Zjevně stále dostáváme stejně dobré přípustné řešení. Když už takovou náhradu nelze provést, máme vybráno několik nejmenších pluků počítače. Nyní ještě analogicky „zvětšíme“ vybrané pluky hráče a jsme hotovi.

Předchozí dvě pozorování dohromady nám postačí k návrhu triviálního kvadratického řešení – jednoduše postupně pro každé K zkontrolujeme, zda dostaneme přípustné řešení, když K největším pluků hráče přiřadíme (ve stejném pořadí) K nejmenších pluků počítače.

Existuje však ještě lepší řešení. Stejně jako v předchozím případě si uvědomíme, že můžeme vybrat několik největších pluků hráče. Postupně (počínaje největším plukem hráče) každému z nich přiřadíme co největší pluk počítače, který ještě dokáže porazit. Zbývá zodpovědět dvě otázky.

První z nich je, proč to funguje. Vezmeme libovolné optimální řešení, do něhož jsme vybrali několik největších pluků hráče a v němž jsou pluky obou hráčů uspořádány podle velikosti. Postupně od největšího budeme procházet pluky počítače a každý z nich zkusíme nahradit *větším* nepoužitým, pokud je to možné. Takto zjevně dostaneme stejně dobré přípustné řešení – a lehce nahlédneme, že právě toto řešení najde i výše popsaný algoritmus.

Druhou otázkou je, jak uvedený postup efektivně implementovat. K tomu si stačí uvědomit, že každému dalšímu pluku hráče přiřadíme menší pluk počítače než předcházejícímu. Proto stačí udržovat si index posledního přiřazeného pluku počítače. Takto najdeme optimální přiřazení v lineárním čase. Nesmíme ale zapomenout, že jsme museli nejprve pluky uspořádat podle velikosti. Proto celková časová složitost tohoto řešení je $O(N \log N)$.

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

int N;                      // počet pluků
int A[10000], B[10000];     // počty vojáků
```

```
// porovnávací funkce pro hodnoty typu int
int icmp(int *a, int *b) { return (*b) - (*a); }

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

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

  // utřídíme vstup
  qsort(A,N,sizeof(int),icmp);
  qsort(B,N,sizeof(int),icmp);

  // najdeme optimální přiřazení
  H = 0; C = 0;
  while (1) {
    // najdeme první pluk počítače, který porazíme
    while (A[H]<=B[C] && C<N) C++;
    // pokud takový neexistuje, skončíme
    if (C==N) break;
    // jdeme najít pár dalšímu pluku hráče
    H++; C++;
  }

  // vypíšeme, kolika plukům hráče jsme našli dvojici
  printf("%d\n",H);
  return 0;
}
```

P-I-2 Teleport

Situaci ze zadání úlohy si můžeme představit jako ohodnocený orientovaný graf. Vrcholy grafu budou představovat lokality, hrany reprezentují teleporty mezi lokalitami. Každá hrana je ohodnocena číslem, které určuje posun v čase při průchodu danou hranou (budeme ho nazývat délka hrany). Úkolem je najít sled♣ z vrcholu 1 do vrcholu N s nejmenším součtem ohodnocení hran (nazveme ho nejkratší), případně vypsat zprávu, že takový sled neexistuje.

Nejprve si všimneme, že pokud mezi dvěma vrcholy vede více hran (stejným směrem), stačí uvažovat jenom tu s nejmenší délkou (jinak bychom dokázali sled zkrátit výměnou hrany za kratší). Dále si všimneme, že náš graf může obsahovat i hrany se zápornou délkou. Mohou tedy nastat dvě situace, kdy hledaný nejkratší sled neexistuje. Buď se z 1 do N po hranách grafu nemůžeme vůbec dostat, nebo existuje sled z vrcholu 1 do vrcholu N takový, že obsahuje cyklus se záporným součtem délek hran. (Po takovém cyklu pak můžeme chodit pořád dokola a stále snižovat celkovou délku sledu. Proto neexistuje nejkratší sled – ke každému sledu totiž dokážeme nalézt kratší.)

Řešení 1, Floyd-Warshallův algoritmus. Graf si uložíme do dvojrozměrného pole `G`, přičemž `G[i][j]` bude délka hrany z vrcholu i do vrcholu j (nebo ∞, jestliže taková hrana neexistuje). Algoritmus vypadá následovně:

```
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 doběhnutí algoritmu je hodnotou `G[i][j]` délka nejkratšího sledu z i do j (případně ∞, pokud žádný neexistuje). Navíc, jestliže `G[i][i]` je záporné pro nějaké i, pak vrchol i leží na nějakém záporném cyklu. Pokud tento vrchol leží na nějakém sledu z 1 do N (tedy `G[1][i]` a `G[i][N]` nejsou rovny ∞), potom existuje libovolně krátký sled z 1 do N.

Idea algoritmu spočívá v dynamickém programování. Když vnější cyklus proběhl k-krát, tak `G[i][j]` je délka nejkratšího sledu z i do j takového, že jako vnitřní vrcholy používá jen vrcholy z množiny $\{1, \ldots, k\}$. Časová složitost tohoto algoritmu je $O(N^3)$, paměťová $O(N^2)$.

Řešení 2, Bellman-Fordův algoritmus. Pro potřeby tohoto algoritmu si budeme hrany grafu uchovávat jednoduše ve třech polích, kde `a[i]` je začátek, `b[i]` je konec a `t[i]` je délka i-té hrany. Idea algoritmu spočívá rovněž v dynamickém programování. Nechť `D[l][i]` je délka takového nejkratšího sledu z 1 do i, který používá právě l hran. Zřejmě `D[0][i]` je ∞ pro všechna i kromě $i = 1$, pro které je to 0. Předpokládejme, že známe `D[l-1][i]` pro všechna i a pro nějaké $l > 0$. Snadno potom spočítáme `D[l][i]` pro libovolné i. V nejkratším sledu používajícím l hran je nějaká hrana poslední a zbytek je

♣ Sled je posloupnost vrcholů $v_1, \ldots, v_k$ taková že mezi v_i a v_{i+1} vede hrana.

nejkratší sled používající $l-1$ hran, který končí v počátku l-té hrany. Stačí jednoduše vyzkoušet všechny možnosti pro tuto poslední hranu. Máme tedy:

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

Při zvolené reprezentaci grafu se výpočet nejsnáze provádí tak, že projdeme postupně všechny hrany a počítáme jednotlivá minima pro všechny vrcholy současně. Víme, že pokud existuje nejkratší sled z 1 do N, bude mít nejvýše $N-1$ hran (neboť neobsahuje cyklus). Výsledkem je tedy minimum z `D[l][N]` pro $l = 1, \ldots, N-1$. Je-li tato hodnota ∞, potom žádný sled neexistuje. Ještě potřebujeme ověřit, zda se nemůžeme dostat do záporného cyklu. Takový cyklus může obsahovat nejvýše N hran (představme si graf s hranami (1,2,1), (2,3,1), ..., ($N-1$,N,1), (N,1,$-N$)). Jestliže tedy existuje sled obsahující záporný cyklus, pak existuje i sled délky nejvýše $2N-1$, který tento cyklus obsahuje. Úpravu vzdáleností tedy spustíme ještě N krát. Když minimum z `D[l][N]` pro $N \le l \le 2N-1$ je menší než výsledek, který jsme našli předtím, potom skutečně existuje sled z 1 do N, který obsahuje záporný cyklus.

Na závěr ještě jedno zjednodušení. Uvědomme si, že nás nezajímají přesné délky sledů s právě k hranami. Proto nám stačí jednorozměrné pole `D[i]` a jednotlivé úpravy stačí provádět jen na něm. Rozmyslete si, že výsledek to nezmění (i když jednotlivé iterace budou vypadat jinak).

Časová složitost tohoto algoritmu je $O(NM)$, paměťová $O(N+M)$.

Poznámka pro zkušené: Pokud znáte Dijkstrův algoritmus, jistě víte, že je rychlejší než oba zde uvedené algoritmy, ale nefunguje na grafech se zápornými hranami. Zkuste si nalézt protipříklad.

Poznámka pro ještě zkušenější: Pokud bychom hledali nejkratší cestu (tedy sled, v němž se vrcholy nemohou opakovat), zadaný úkol by se již stal NP-těžkým problémem (všechny známé polynomiální algoritmy na hledání nejkratší cesty totiž fungují jen pro speciální typy grafů, většinou jde o grafy bez záporných cyklů).

```
program teleport;
const INF = 1000000000;
      MAXN = 1000;
      MAXM = 50000;
var N, M: integer;
    a: array[1..MAXM] of integer;
    b: array[1..MAXM] of integer;
    t: array[1..MAXM] of integer;
    D: array[1..MAXN] of longint;
    i, l, k: integer;
    vysledek: longint;

procedure nacti;
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 pocitej;
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];
  vysledek:=D[N];

  if vysledek=INF then
    writeln(f,'Vědci umřou hlady')
```

```
  else begin
    for l:=1 to N 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];

    if D[N]<vysledek then
      writeln(f,'Vědci poznají vznik vesmíru')
    else
      writeln(f,vysledek);
  end;

  close(f);
end;

begin
  nacti;
  pocitej;
end.
```

P-I-3 Posádky

Při řešení této úlohy použijeme postup známý jako *dynamické programování* – budeme řešit úlohu ze zadání (a její mírně změněné verze) postupně pro různé části sítě cest, přičemž z výsledků pro menší části budeme počítat výsledky pro větší části mapy.

Soustavu cest v království budeme reprezentovat grafem – vrcholy grafu představují města, hrany cesty mezi nimi. Ze zadání víme, že tento graf je strom, tzn. má N vrcholů, právě $N-1$ hran a je souvislý. Libovolný z vrcholů stromu nazveme kořenem stromu. Všem jeho sousedním vrcholům budeme říkat synové kořene, jejich ostatní sousedi budou zase jejich syny, atd. Můžeme si to celé představit tak, že celý strom zavěsíme za kořen. Otcem vrcholu je ten jeho soused, který je nad ním, ostatní sousedi vrcholu budou jeho synové. Nadále budeme pojmem *podstrom s kořenem v* rozumět tu část našeho stromu, z níž se do kořene můžeme dostat jedině přes vrchol v. Když budeme mluvit o *podstromu*, myslíme tím podstrom s libovolným kořenem♣. Název *skupina posádek* bude označovat množinu vrcholů s posádkami, která je souvislá a nesousedí s žádným dalším vrcholem s posádkou.

Všimněme si, jak vypadá optimální řešení. Pro kořen máme dvě možnosti: buď tam posádka je, nebo tam není. Pokud tam není, zbylo nám několik *samostatných* podstromů a v každém z nich jsou posádky rozmístěny optimálním způsobem. A co když v kořeni posádka je? Uvažujme skupinu posádek obsahující kořen. V žádném z vrcholů, které s ní sousedí, posádka být nemůže. Když odstraníme všechny tyto vrcholy z grafu, opět nám zbude několik *samostatných* podstromů. A opět v každém z těchto podstromů budou posádky rozmístěny optimálně. Kdybychom tedy znali optimální způsoby rozmístění posádek pro všechny podstromy, dokázali bychom vyzkoušením konečného počtu možností najít optimální způsob rozmístění posádek pro celý strom. Jenomže každý z podstromů je sám rovněž stromem a můžeme pro něj zopakovat celou tuto úvahu.

Princip *dynamického programování* spočívá v tom, že se na problém podíváme z opačné strany. Pro *listy* (vrcholy, které nemají žádného syna) je řešení triviální. Začneme tedy od listů a postupně budeme zpracovávat čím dál tím větší podstromy, dokud nenajdeme optimální řešení pro celý strom. Uvědomte si, že v okamžiku, když zpracováváme nějaký podstrom, byly už všechny *jeho* podstromy zpracovány a známe tedy již jejich optimální řešení.

Aby se nám řešení snadněji implementovalo, provedeme následující úvahu:

Nechť $A_{v,i}$ je hodnota nejlepšího řešení pro podstrom s kořenem v, jestliže víme, že skupina posádek obsahující vrchol v má velikost i. (Přitom i je od 0 do 3, $i=0$ znamená, že ve vrcholu v není posádka.) Dále nechť $A_v = \max A_{v,i}$ je hodnota nejlepšího řešení pro podstrom s kořenem v. Připomeňme si ze zadání, že číslo b_v udává, kolik přispěje posádka ve vrcholu v k bezpečnosti království. Ukážeme si, jak lze lehce spočítat hodnotu A_v pro nějaký vrchol v, jestliže již budeme znát tuto hodnotu pro všechny jeho syny. Nechť v má syny $s_1, \ldots, s_K$.

- Zjevně $A_{v,0} = \sum_{i=1}^{K} A_{s_i}$ – každý z podstromů vyřešíme optimálně.
- Dále $A_{v,1} = b_v + \sum_{i=1}^{K} A_{s_i,0}$ – započítáme přínos vrcholu v a každý jeho podstrom vyřešíme optimálně s tím, že jeho vrchol musí zůstat prázdný.
- Podobně spočítáme $A_{v,2}$ (jenom v jednom podstromu použijeme $A_{s_i,1}$) a $A_{v,3}$ (ve dvou podstromech zvolíme $A_{s_i,1}$ nebo v jednom $A_{s_i,2}$).
- Nakonec ještě A_v, které spočteme jako maximum z hodnot $A_{v,0}$ až $A_{v,3}$.

Čas potřebný na zpracování jednoho vrcholu je úměrný počtu jeho sousedů, celkový čas je tedy úměrný počtu všech vrcholů a hran dohromady. Hran je však pouze $N-1$, a proto je celková časová složitost rovna $O(N)$.

♣ Často se podstromem rozumí libovolný podgraf, který je stromem. Zcela korektně bychom měli naše podstromy nazývat například *podstrom indukovaný vrcholem v*. Čtenář jistě pochopí, že kvůli snadnějšímu vyjadřování jsme raději zvolili tuto dohodu.

```
#include <stdio.h>
#define MAX_N 10000
#define MAX(a,b) (((a) > (b)) ? (a) : (b))

int N;
int s[MAX_N];                                   /* sousedi vrcholů */
int h[MAX_N+1];                                 /* h[i]..h[i+1] jsou indexy
                                                   sousedů vrcholu i v poli s */
int b[MAX_N];                                   /* ceny vrcholů */
struct hrana { int f,t; } pom[MAX_N];           /* pomocné pole pro načtení hran */

int A[MAX_N][4];
int nej[MAX_N][5];
#define NEJ_CENA  0                             /* nej[i][0] je max{A[i][k]|k={0,1,2,3}} */
#define NEJ_INDEX 1                             /* nej[i][1] je to k, které se pro nej[i][0] použije
                                                   0 - u všech synů se použije jejich max
                                                   1 - u všech synů se použije A[syn][0]
                                                   2 - jako 1, ale u syna p se použije A[p][1]
                                                   3 - jako 2, ale u syna q se použije A[q][1]
                                                   4 - jako 1, ale u syna r se použije A[r][2] */
#define NEJ_P     2
#define NEJ_Q     3
#define NEJ_R     4

void nacti(void) {
  int i;

  scanf("%d",&N);
  for (i=0;i<N-1;i++) {
    int a,b;
    scanf("%d%d",&a,&b);
    pom[i].f=--a;                               /* přečíslujeme od nuly */
    pom[i].t=--b;
    h[a]++;
    h[b]++;
  }

  for (i=0;i<N;i++) scanf("%d",b+i);

  for (i=0;i<N;i++) h[i+1]+=h[i];               /* ještě vyplníme s a v načtenými hranami */
  for (i=0;i<N-1;i++) {
    int a=pom[i].f,b=pom[i].t;
    s[--h[a]]=b;
    s[--h[b]]=a;
  }
}

void zpracuj(int v,int o) {                     /* urči nejlepší cenu v s otcem o */
  int syn,i,c;
  int nejA2cena=0,nejA2index=v;                 /* pro syna s nejlepším A[syn][2] */
  int nejA1cena1=0,nejA1index1=v;               /* a pro dva syny s nejlepším A[syn][1] */
  int nejA1cena2=0,nejA1index2=v;

  A[v][0]=0;
  A[v][1]=b[v];
  for (i=h[v];i<h[v+1];i++) if (syn=s[i],syn!=o) {
    zpracuj(syn,v);                             /* zpracuj rekurzí */

    A[v][0]+=nej[syn][NEJ_CENA];                /* uprav A[v][0,1] */
    A[v][1]+=A[syn][0];

    if (c=A[syn][2]-A[syn][0],c > nejA2cena) {  /* A[v][2,3] zatím přímo upravit */
      nejA2cena=c;                              /* nemůžeme, ale připravíme se na to */
      nejA2index=syn;
    }
```

```
    if (c=A[syn][1]-A[syn][0], c <= nejA1cena2) continue; /* to by nám nepomohlo */
    if (c<nejA1cena1) {                          /* je to jenom druhý nejlepší s A[syn][1] */
      nejA1cena2=c;
      nejA1index2=syn;
    } else {                                     /* je to úplně nejlepší syn (zatím :) */
      nejA1cena2=nejA1cena1;
      nejA1index2=nejA1index1;
      nejA1cena1=c;
      nejA1index1=syn;
    }
  }

  A[v][2]=A[v][1]+nejA1cena1;                    /* ještě doupravíme A[v][2,3] a vyplníme nej */
  A[v][3]=A[v][1]+MAX(nejA1cena1 + nejA1cena2,nejA2cena);

  nej[v][NEJ_CENA]=A[v][0];
  nej[v][NEJ_INDEX]=0;
  for (i=1;i<4;i++)
    if (A[v][i]>nej[v][NEJ_CENA]) {
      nej[v][NEJ_CENA]=A[v][i];
      nej[v][NEJ_INDEX]=i;
    }

  if (nej[v][NEJ_INDEX]==3 && nejA2cena > nejA1cena1+nejA1cena2)
    nej[v][NEJ_INDEX]=4;                         /* speciální případ pro syna A[syn][2] */

  nej[v][NEJ_P]=nejA1index1;
  nej[v][NEJ_Q]=nejA1index2;
  nej[v][NEJ_R]=nejA2index;
}

void vypis(int v,int o,int index) {
  int syn,i;

  if (index) printf("%d",v+1);
  for (i=h[v];i<h[v+1];i++)
    if (syn=s[i],syn!=o) {
      switch (index) {
        case 0: vypis(syn,v,nej[syn][NEJ_INDEX]); break;
        case 1: vypis(syn,v,0); break;
        case 2: vypis(syn,v,syn==nej[v][NEJ_P]); break;
        case 3: vypis(syn,v,syn==nej[v][NEJ_P] || syn==nej[v][NEJ_Q]); break;
        case 4: vypis(syn,v,2*(syn==nej[v][NEJ_R])); break;
      }
    }
}

int main(void) {
  nacti();

  zpracuj(0,-1);                                 /* (-1) je fiktivní neexistující otec */

  printf("%d\n",nej[0][NEJ_CENA]);
  vypis(0,-1,nej[0][NEJ_INDEX]);
  putchar('\n');

  return 0;
}
```

P-1-4 Paralelizátor

Soutěžní úloha a. Nechť řetězec *jehla* má délku N a řetězec *seno* má délku M. Ukážeme řešení pracující v čase $O(\log N + \log M)$.

Základní myšlenka: Kdyby nám někdo řekl, kde začíná nějaký výskyt řetězce *jehla* v řetězci *seno*, pomocí volání **Both** snadno ověříme v čase $O(\log N)$, zda má pravdu. Podobně jako jsme v příkladu uvedeném v zadaní dokázali paralelně ověřit všechny dělitele čísla, můžeme zde paralelně porovnat N dvojic písmen.

Když nám ale nikdo neřekne, kde začíná nějaký výskyt řetězce *jehla*, pomocí **Some** paralelně ověříme všechny možné začátky a úspěšně skončíme, jestliže některý z nich vyhovuje.

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

{ forall() paralelně spustí N kopií, v nichž cislo = 0..(N-1),
  úspěšně skončí, jestliže všechny úspěšně skončí }
procedure forall(var cislo: integer, N: integer);
var moc2, cifer, i, x: integer;
begin
   { zjistíme, kolik má číslo N-1 cifer ve dvojkové soustavě }
   moc2:= 1;
   cifer:= 0;
   while (moc2 <= N-1) do begin moc2:= moc2 * 2; inc(cifer); end;
   { vygenerujeme čísla od 0 do 2^cifer - 1 }
   cislo:= 0;
   for i:=1 to cifer do begin Both(x); cislo:= 2*cislo + x; end;
   if (cislo >= N) then Accept;
end;

{ exists() paralelně spustí N kopií, v nichž cislo = 0..(N-1),
  úspěšně skončí, jestliže některá z nich úspěšně skončí }
procedure exists(var cislo: integer, N: integer);
var moc2, cifer, i, x :integer;
begin
   { zjistíme, kolik má číslo N-1 cifer ve dvojkové soustavě }
   moc2:= 1;
   cifer:= 0;
   while (moc2 <= N-1) do begin moc2:= moc2 * 2; inc(cifer); end;
   { vygenerujeme čísla od 0 do 2^cifer - 1 }
   cislo:= 0;
   for i:=1 to cifer do begin Some(x); cislo:= 2*cislo + x; end;
   if (cislo >= N) then Reject;
end;

var N, M: integer;
    zacatek, pozice: integer;

begin
   { zjistíme délky řetězců }
   N:= length(jehla);
   M:= length(seno);
   if (M < N) then Reject;

   { zkusíme, zda existuje možný začátek }
   exists(zacatek, M-N+1);
   { paralelně se podíváme na všechny pozice řetězce jehla }
   forall(pozice, N);
   { ověříme, zda jsou odpovídající si písmena stejná }
   if (jehla[pozice+1] = seno[zacatek+pozice+1]) then Accept;
   Reject;
end.
```

Soutěžní úloha b. Přímočarým řešením je vypočítat postupně všechna políčka pyramidy a hodnotu horního porovnat s V. Políček je $N(N-1)/2$, proto takové řešení počítá v čase $O(N^2)$.

Zatím jsme ale nevyužili dvě skutečnosti: to, že známe hodnotu V – a samozřejmě sílu paralelizátoru. Ukážeme si nyní řešení, které počítá v čase $O(N)$.

Kdybychom znali obě čísla v druhém řádku shora, dokázali bychom snadno ověřit, zda je V správným součtem celé pyramidy – jednoduše bychom je sečetli. My je však neznáme – jenom víme, že jsou z rozmezí 0 a 9 999 včetně. Pomocí příkazu **Some** vyzkoušíme všechny dvojice takovýchto čísel, které (modulo 10 000) dávají součet V – tj. zkusíme „uhodnout" ta správná čísla, která tam mají být. Pro každou ze zkoušených možností potřebujeme o obou nových číslech ověřit, zda patří na to místo, kam jsme je právě umístili. Pomocí jednoho volání **Both** ověříme obě čísla najednou. No a už si stačí jen uvědomit, že jsme dostali opět přesně původní úlohu, jenom s pyramidou, která má o 1 kratší základnu.

Když se takto dostaneme až ke spodnímu řádku pyramidy, místo hádání se už jen jednoduše podíváme na čísla ze vstupu.

Proč naše řešení funguje? Uvědomme si, že vlastně postupujeme pyramidou „shora dolů" a zkoušíme ji všemi způsoby doplnit. Úspěšně skončíme tehdy, když se nám podaří doplnit celou pyramidu. No a jelikož pyramida je svým spodním řádkem jednoznačně určená, toto se nám podaří jedině tehdy, když je skutečně V na jejím vrcholu.

„Zpracovat" jednu úroveň pyramidy dokážeme v konstantním čase, úrovní je $O(N)$, proto i časová složitost našeho programu je $O(N)$.

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

{ exists() paralelně spustí N kopií, v nichž cislo = 0..(N-1),
  úspěšně skončí, jestliže některá z nich úspěšně skončí }
procedure exists(var cislo: integer, N: integer);
var moc2, cifer, i, x :integer;
begin
   { zjistíme, kolik má číslo N-1 cifer ve dvojkové soustavě }
   moc2:= 1;
   cifer:= 0;
   while (moc2 <= N-1) do begin moc2:= moc2 * 2; inc(cifer); end;
   { vygenerujeme čísla od 0 do 2^cifer - 1 }
   cislo:= 0;
   for i:=1 to cifer do begin Some(x); cislo:= 2*cislo + x; end;
   if (cislo >= N) then Reject;
end;

{ Over() ověří, zda na políčku ["radek","sloupec"] je číslo "hodnota"
  řádky číslujeme zdola, sloupce zleva, oboje od 1 }
procedure Over(radek, sloupec, hodnota: integer);
var leve, prave, x: integer;
begin
   { když už jsme dole, ověříme snadno }
   if (radek = 1) then
      if (hodnota = A[sloupec]) then Accept else Reject;

   { zkusíme všechny možnosti pro levé pod ním }
   exists(leve, 10000);
   { dopočítáme pravé pod ním }
   prave:= hodnota - leve;
   if (prave < 0) then Inc(prave, 10000);

   { ověříme, zda jsme obě čísla vybrali správně }
   Both(x);
   if (x=0) then
      Over(radek-1, sloupec, leve)
   else
      Over(radek-1, sloupec+1, prave);
end.

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