MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

51. ročník, školský rok 2001/2002

Riešenia úloh 3. kola kategórie P

1. súťažný deň

P-III-1

Najprv zaveďme niekoľko označení. Označme X_i i-ty stĺpec v matici X. Prvok v i-tom riadku a j-tom stĺpci označíme $x_{i,j}$. Nech v i-tom riadku matice X tvorí prvých j prvkov postupnosť, do ktorej sa dá za žolíky dosadiť tak, aby bola neklesajúca. Potom *signatúrou* prvku $x_{i,j}$ v tejto matici budeme nazývať najmenšie číslo, ktoré môžeme umiestniť v riadku i do $(j+1)$. stĺpca tak, aby nebola porušená podmienka neklesajúcosti. Signatúra prvku $x_{i,j}$ je teda najväčšie (=posledné) číslo z tých prvkov $x_{i,1}, x_{i,2}, \ldots, x_{i,j}$, ktoré nie sú žolíkmi, alebo 0, ak všetky tieto prvky sú žolíkmi. Signatúra stĺpca je *utriedená*, ak postupnosť signatúr prvkov od prvého riadku po posledný je neklesajúca.

Predpokladajme teraz, že máme dané stĺpce $A'_1, A'_2, \ldots, A'_k$, kde každý stĺpec A'_i je preusporiadaním stĺpca A_i. Maticu A^* budeme nazývať *rozšírením* stĺpcov $A'_1, A'_2, \ldots, A'_k$, ak

- matica A^* je riešením našej úlohy (t.j. každý stĺpec A^*_i je preusporiadaním stĺpca A_i a v matici A^* možno nahradiť žolíky číslami tak, aby každý riadok bol neklesajúci),
- prvých k stĺpcov matice A^* sa zhoduje so stĺpcami $A'_1, A'_2, \ldots, A'_k$.

Predpokladajme, že už sme našli prvých k stĺpcov riešenia $A'_1, A'_2, \ldots, A'_k$, a že signatúra stĺpca A'_k je utriedená. Ukážeme teraz, ako preusporiadať ďalší stĺpec.

Zotriedime všetky nežolíkové prvky v stĺpci A_{k+1} od najmenšieho po najväčší a potom postupne skúmame políčka stĺpca oddola nahor. Na políčko $a'_{i,k+1}$ uložíme najväčšie doteraz nepoužité číslo v stĺpci, ak je aspoň tak veľké ako signatúra $a'_{i,k}$. V opačnom prípade tam položíme žolík a ak žiaden žolík nemáme, vyhlásime, že matica sa nedá usporiadať.

Všimnime si, že signatúra stĺpca A'_{k+1} je tiež utriedená. Preto tento algoritmus môžeme použiť na postupné vygenerovanie všetkých stĺpcov matice.

Dôkaz správnosti. Je zrejmé, že ak sa nášmu algoritmu podarí preusporiadať všetky stĺpce matice, tak dostane maticu, ktorá má všetky riadky neklesajúce (t.j. nájde správne riešenie). Zostáva dokázať, že ak správne riešenie existuje, tak algoritmus ho aj nájde, t.j. neskončí neúspechom na niektorom stĺpci. Najskôr dokážeme dve pomocné tvrdenia.

Tvrdenie 1. Sú dané stĺpce $A'_1, A'_2, \ldots, A'_k$, kde každý stĺpec A'_i je preusporiadaním stĺpca A_i a signatúra stĺpca A'_k je utriedená. Označme A'_{k+1} preusporiadanie stĺpca A_{k+1}, ktoré dostaneme našim algoritmom. Potom, ak existuje rozšírenie stĺpcov $A'_1, \ldots, A'_k$, tak existuje aj rozšírenie stĺpcov $A'_1, \ldots, A'_{k+1}$.

Dôkaz. Predpokladajme, že existuje rozšírenie A^* stĺpcov $A'_1, \ldots, A'_k$. Môžu nastať dva prípady. Ak sa stĺpec A^*_{k+1} zhoduje so stĺpcom A'_{k+1}, potom matica A^* je rozšírením aj stĺpcov $A'_1, \ldots, A'_{k+1}$, a teda naše tvrdenie automaticky platí. Predpokladajme teda, že stĺpce A^*_{k+1} a A'_{k+1} sa nezhodujú.

Vezmime najspodnejší riadok, v ktorom sa tieto stĺpce líšia; označme tento riadok j. Môžu nastať dva prípady:

- $a'_{j,k+1} = *$ ($a^*_{j,k+1}$ je teda číslo). Číslo $a^*_{j,k+1}$ musí vyskytovať niekde vyššie v stĺpci A'_{k+1} a je ho možné pridať namiesto žolíka v j-tom riadku. To je ale v rozpore s našim algoritmom, ktorý používa žolíky iba v prípade, že nemôže použiť číslo. Tento prípad teda nemôže nastať.

- $a'_{j,k+1}$ je číslo ($a^*_{j,k+1}$ je buď iné číslo, alebo žolík). Stĺpec A^*_{k+1} musí obsahovať číslo $a'_{j,k+1}$ na nejakom inom riadku i, pričom $i < j$, lebo stĺpce A'_{k+1} a A^*_{k+1} sa zhodujú na riadkoch $j+1, \ldots, m$.

  Vymeňme riadky i a j v stĺpcoch $k+1, \ldots, n$ v matici A^*; nazvime výslednú maticu A^{**}. Dokážeme, že matica A^{**} je riešením nášho problému. Jediné miesto, kde by mohla byť porušená podmienka, že riadky sú neklesajúce, je v riadku j za prvkom $a^{**}_{j,k}$, alebo v riadku i za prvkom $a^{**}_{i,k}$. V prípade riadku j, $a^{**}_{j,k+1} = a^*_{i,k+1} = a'_{j,k+1}$ a podmienka teda nie je porušená, lebo prvok $a'_{j,k+1}$ použije v riadku j aj náš algoritmus.

  Rozoberme teraz situáciu na riadku i. Označme si $w = a^{**}_{i,k+1} = a^*_{j,k+1}$. Všimnime si, že signatúra prvku $a^{**}_{j,k}$ je väčšia alebo rovná ako signatúra prvku $a^{**}_{i,k}$ (z predpokladu, že signatúra stĺpca k je utriedená). Ak prvok w je číslo a bolo ho možné použiť na riadku j, je ho možné použiť aj na riadku i. Ak prvok w je žolík, tak jeho signatúra sa premiestnením do riadku i nezväčšila, preto zvyšok riadku možno použiť bezo zmeny.

Ak sa stĺpce A^{**}_{k+1} a A'_{k+1} zhodujú, tvrdenie je dokázané. V opačnom prípade použijeme ten istý argument znova na maticu A^{**}, až kým sa stĺpce nebudú zhodovať (v každej iterácii odstránime jednu nezhodu). □

Tvrdenie 2. Sú dané stĺpce $A'_1, A'_2, \ldots, A'_k$, kde každý stĺpec A'_i je preusporiadaním stĺpca A_i a signatúra stĺpca A'_k je utriedená. Ak náš algoritmus pri generovaní stĺpca A'_{k+1} vyhlási, že maticu nemožno preusporiadať, tak neexistuje doplnenie stĺpcov $A'_1, A'_2, \ldots, A'_k$.

Dôkaz. Nech sa náš algoritmus zastavil na riadku j. To znamená, že nám nezostali žiadne žolíky, a navyše signatúra $a'_{j,k}$ je väčšia, ako ľubovoľný zo zostávajúcich prvkov. Označme túto množinu zostávajúcich čísel Z.

Predpokladajme teraz, že existuje rozšírenie A^* stĺpcov $A'_1, A'_2, \ldots, A'_k$. Rovnakým postupom, ako v predchádzajúcom dôkaze, možno nájsť rozšírenie A^{**}, ktoré je totožné s čiastočným riešením vygenerovaným naším algoritmom (tzn. $a^{**}_{i,k+1} = a'_{i,k+1}$ pre $j < i \leq m$). Vezmime si prvok $a^{**}_{j,k+1}$. Určite to nie je žolík, pretože všetky žolíky sme už minuli v nižších riadkoch. Preto to musí byť číslo z množiny Z. Tieto čísla sú ale všetky menšie, ako signatúra prvku $a'_{j,k}$, preto A^{**} nemôže byť rozšírením $A'_1, A'_2, \ldots, A'_k$ a teda ani A^* nie je rozšírením. To je ale spor s predpokladom. □

Z týchto dvoch tvrdení už ľahko dokážeme správnosť celého algoritmu. Predpokladajme, že maticu A možno preusporiadať. Potom pre 0 stĺpcov existuje rozšírenie týchto stĺpcov. Navyše, ak vieme nájsť preusporiadanie k stĺpcov, pre ktoré existuje rozšírenie, tak podľa Tvrdenia 2 náš algoritmus nájde preusporiadanie $(k+1)$-vého stĺpca. Podľa Tvrdenia 1 potom existuje rozšírenie takéhoto preusporiadania prvých $k+1$ stĺpcov. Algoritmus teda môžeme zopakovať n krát a dostaneme celé riešenie.

Časová a pamäťová zložitosť. V každom stĺpci potrebujeme utriediť čísla, čo je možné urobiť v čase $O(m \log m)$. Zvyšné operácie sa dajú spraviť v čase $O(m)$ pre každý stĺpec. Celkový čas výpočtu teda je $O(nm \log m)$. Pamäťová zložitosť je $O(mn)$.

program *P_III_1*;
var a : **array** [1..100, 1..100] **of** *integer*;

```
        { Matica A. Predpokladáme, že žolíky sú reprezentované ako -1 }
    n, m : integer; { Rozmery matice A }
    sig : array [1..100] of integer; { Signatúra posledného stĺpca }
    stlpec : array [1..100] of integer; { Pomocné pole na jeden stĺpec }
procedure nacitaj;
var i, j : integer;
begin
  readln(m, n);   { načítaj rozmery }
  for i := 1 to m do   { načítaj maticu }
    for j := 1 to n do
      read(a[i,j]);
end; { nacitaj }

procedure tried(k : integer);
{ Utriedi k-ty stĺpec matice. Pre úsporu miesta použijeme
  kvadratické triedenie, možno nahradiť n log n triedením.
  Žolíky (reprezentované -1) budú navrchu. }
var i, j, min, pom : integer;
begin
  for i := 1 to m - 1 do begin
    min := i;    { nájdi minimum v neutriedenej časti stĺpca }
    for j := i + 1 to m do begin
      if a[j,k] < a[min,k] then min := j;
    end;
    { vymeň a[i,k] a a[min,k] }
    pom := a[i,k]; a[i,k] := a[min,k]; a[min,k] := pom;
  end;
end; { tried }

procedure vypis;
var i, j : integer;
begin
  for i := 1 to m do begin
    for j := 1 to n do
      write(' ',a[i][j]);
    writeln;   { ukonči riadok i }
  end;
end;

var i, k, pouzi : integer;
begin  { hlavný program }
  nacitaj;    { vstup }
  for i := 1 to m do sig[i] := 0; { inicializuj signatúru }
  for k := 1 to n do begin { pre všetky stĺpce }
    tried(k);   { utrieď stĺpec }
    pouzi := m;   { posledné nepoužité číslo }
    { ukladaj čísla odspodu do pomocného poľa "stlpec" }
    for i := m downto 1 do begin
      if a[pouzi][k] >= sig[i] then begin
```

```
            stlpec[i] := a[pouzi][k];
            pouzi := pouzi - 1;
          end
          else stlpec[i] := -1; { ak sa nedá použiť číslo, použi žolík }
        end;
        { ak zostali nepoužité čísla, nemožno usporiadať }
        if (pouzi >= 1) and (a[pouzi][k] <> -1) then begin
          writeln('Maticu nemozno usporiadat');
          exit;
        end;
        { skopíruj pomocné pole späť do matice a prepočítaj signatúru }
        for i := 1 to m do begin
          a[i][k] := stlpec[i];
          if stlpec[i] <> -1 then sig[i] := stlpec[i]
        end;
      end;
      vypis; { výpis matice }
    end.
```

P-III-2

Úlohu je možné riešiť dynamickým programovaním. Trasu, ktorá spĺňa všetky podmienky vyhovujúcej trasy, okrem toho, že nemusí končiť v najnižšom orientačnom bode budeme nazývať *čiastočná trasa*. Naše riešenie bude počítať počty čiastočných trás s rôznymi koncami využívajúc informáciu spočítanú predtým.

V prvom kroku zotriedime body podľa y-ovej súradnice od najvyššie položeného po najnižšie položený. Očíslujme body v tomto utriedenom poradí číslami od 1 po N. Pre i, j ($1 \leq i < j \leq N$) nech $a[i,j]$ je počet čiastočných trás, ktoré majú i ako predposledný orientačný bod a j ako posledný. Pre $i = 1$ máme $a[i,j] = 1$, lebo každá čiastočná trasa začína v bode 1. Predpokladajme teraz, že sme už spočítali hodnoty $a[i',j']$ pre všetky i', j', kde $i' < i$ a chceme spočítať $a[i,j]$. Ak má trasa končiť úsekom (i,j), musí do bodu i ísť z nejakého bodu k, pre ktorý platí, že uhol otočenia zo smeru (k,i) do smeru (i,j) je menej ako 45° a bod k je vyššie položený ako bod i. Nech $S(i,j)$ je množina všetkých bodov k s týmito vlastnosťami. Keďže každé $k \in S(i,j)$ je menšie ako i, máme už spočítanú hodnotu $a[k,i]$. Hodnota $a[i,j]$ sa spočíta jednoducho ako súčet $a[k,i]$ pre všetky k z množiny $S(i,j)$, t.j.

$$a[i,j] = \sum_{k \in S(i,j)} a[k,i].$$

Na základe tohoto vzťahu ľahko zostavíme algoritmus pracujúci v čase $O(N^3)$. Počiatočné triedenie trvá $O(N \log N)$. Existuje kvadratický počet dvojíc i, j, pre ktoré chceme spočítať hodnotu $a[i,j]$. Pre každú takú dvojicu musíme najprv nájsť množinu $S(i,j)$, čo sa dá spraviť v čase $O(N)$ tak, že pre každý bod k spočítame príslušný bod otočenia a zistíme, či je menší ako 45°. Keď nájdeme množinu $S(i,j)$, sčítame hodnoty $a[k,i]$ pre všetky $k \in S(i,j)$. To tiež trvá $O(N)$. Spolu teda potrebujeme čas $O(N^3)$ na spočítanie celej tabuľky a. Výsledný počet vyhovujúcich trás je súčet hodnôt $a[i,N]$ pre všetky $1 \leq i < N$ (t.j. počet všetkých čiastočných trás končiacich v bode N). Tento súčet tiež ľahko nájdeme v čase $O(N)$. Dostali sme teda algoritmus pracujúci v čase $O(N^3)$.

Tento algoritmus sa dá ďalej zrýchliť použitím podobných techník ako v predchádzajúcich kolách. Trik spočíva v tom, že pre dané i budeme počítať hodnoty $a[i,j]$ pre všetky j ($j > i$)

naraz. Najskôr zotriedime všetky body proti smeru hodinových ručičiek okolo bodu i. Pre každý bod j ($j > i$) množina $S(i,j)$ tvorí súvislý úsek v takto utriedenom zozname. Nech $l(i,j)$ je najľavejší bod tohto úseku a nech $p(i,j)$ je najpravejší bod. Teraz si predstavme, že bod j otáčame proti smeru hodinových ručičiek okolo bodu i. Množina $S(i,j)$ sa bude meniť, a to tak, že body $l(i,j)$ ani $p(i,j)$ sa tiež budú otáčať proti smeru hodinových ručičiek. V našom algoritme nebudeme otáčať bod j, ale budeme uvažovať rôzne orientačné body $j > i$ v poradí, v akom sú v zozname utriedenom okolo bodu i. Budeme udržovať dva indexy $l(i,j)$ a $p(i,j)$, ktoré zakaždým zvyšujeme, až kým nezodpovedajú práve skúmanému bodu j. Súčasne budeme udržovať aj súčet hodnôt $a[k,i]$ pre k z úseku medzi $l(i,j)$ a $p(i,j)$. Vždy keď zvýšime $l(i,j)$, znížime tento súčet o hodnotu $a[l(i,j),i]$, ktorá sa práve dostala mimo úseku. Podobne, ak zvýšime hodnotu $p(i,j)$, zvýšime súčet o hodnotu $a[p(i,j),i]$, ktorá sa práve dostala do vnútra úseku.

Pre každú hodnotu j teda najprv posunieme indexy $l(i,j)$ a $p(i,j)$ na ich patričné miesto a upravujeme pritom súčet a nakoniec tento súčet uložíme v $a[i,j]$. Celkovo pre všetky hodnoty j ($j > i$) index $l(i,j)$ prejde cez každý orientačný bod k ($k < i$) nanajvýš raz a podobe každý aj index $p(i,j)$. Udržovanie indexov a súčtu teda zaberie $O(N)$ času pre každé i. Samotné ukladanie hodnôt $a[i,j]$ tiež zaberie čas $O(N)$ pre každé i. Triedenie podľa uhla zaberie $O(N \log N)$ času pre každé i. Celkovo teda potrebujeme čas $O(N \log N)$ pre každé i, čiže $O(N^2 \log N)$ spolu (to už zahŕňa aj počiatočné triedenie podľa y-ovej súradnice a spočítanie výsledného počtu vyhovujúcich ciest z hodnôt $a[i,N]$. Pamäťová zložitosť je $O(N^2)$.

Pri implementácii treba dávať pozor na rôzne okrajové prípady, napríklad keď množina $S(i,j)$ je prázdna. Kvôli úspore miesta program uvedený v riešení používa kvadratické triedenie.

```
program P_III_2;
const MAX_N = 100;
var N : integer; { počet bodov }
    x, y : array[1..MAX_N] of real; { súradnice bodov }
    pocet : array[1..MAX_N, 1..MAX_N] of integer;
            { počet čiastočných trás s danou poslednou hranou }
    ind : array[1..MAX_N] of integer;
            { indexové pole na triedenie podľa uhla }
    uhol_ind : array[1..MAX_N] of real;
            { uhol zodpovedajúci bodu v ind[i] }

function uhol(x, y : real) : real;
var vysl : real;
begin { implementácia funkcie "uhol" de novanej v zadaní }
  if x = 0 then begin { špeciálny prípad x=0 }
    if y > 0 then uhol := 90
    else uhol := 270;
  end
  else begin { x<>0, môžeme použit arctan }
    vysl := arctan(y/x) / pi * 180;
    if x < 0 then vysl := vysl - 180; { zmeň výsledok podľa kvadrantu }
    if vysl < 0 then vysl := vysl + 360;
    uhol := vysl;
  end;
end; { uhol }
```

```
procedure tried(stred : integer);
var pom_r : real;
var i, j, min, pom_i : integer;
begin { utriedi body okolo bodu "stred", ich indexy uloží
        do pola "ind" a uhly do pola "uhol_ind". }
   { inicializuj ind a spočítaj uhly }
   j := 1;
   for i := 1 to N do begin
      if i <> stred then begin { bod "stred" vynechaj }
         ind[j] := i;
         uhol_ind[j] := uhol(x[i] - x[stred], y[i] - y[stred]);
         j := j + 1;
      end;
   end;
   { tried ind podla uhol_ind }
   for i := 1 to N - 1 do begin
      min := i;    { nájdi minimum v neutriedenej časti pola }
      for j := i + 1 to N - 1 do begin
         if uhol_ind[j] < uhol_ind[min] then min := j;
      end;
      { ulož minimum do ind[i] }
      pom_i := ind[i]; ind[i] := ind[min]; ind[min] := pom_i;
      pom_r := uhol_ind[i]; uhol_ind[i] := uhol_ind[min];
      uhol_ind[min] := pom_r;
   end;
end; { tried }

procedure spocitaj_z_bodu(stred : integer);
var zac, kon, sucet, i : integer;
   uhol_zac, uhol_kon      : real;
begin { spočítaj pocet[stred,i] pre všetky i>stred }
   tried(stred);
   zac := 1;   { prvý bod, ktorý je aspoň na začiatku výseku }
   kon := 1;   { prvý bod, ktorý je za koncom výseku }
   sucet := 0; { súčet bodov vo výseku }
   for i := stred to N - 1 do begin
      uhol_zac := uhol_ind[i] - 180 - 45;    { spočítaj výsek }
      uhol_kon := uhol_ind[i] - 180 + 45;
      if uhol_kon > 180 then uhol_kon := 180;

      { posuň indexy "zac" a "kon", udržuj "sucet" }
      while uhol_ind[zac] < uhol_zac do begin
         sucet := sucet - pocet[ind[zac], stred];
         zac := zac + 1;
      end;
      while uhol_ind[kon] <= uhol_kon do begin
         sucet := sucet + pocet[ind[kon], stred];
         kon := kon + 1;
      end;
```

```
      pocet[stred, ind[i]] := sucet;  {ulož výsledok}
   end;
end; { spocitaj_z_bodu }

procedure inicializuj;
var i, j  : integer;
   pom : real;
begin { načítaj vstup a strieď podľa y zhora nadol }
   read(N);
   for i := 1 to N do begin
      read(x[i], y[i]);
      j := i;   { vlož nový bod na správne miesto }
      while (j > 1) and (y[j] >= y[j - 1]) do begin
         pom := x[j - 1]; x[j - 1] := x[j]; x[j] := pom;
         pom := y[j - 1]; y[j - 1] := y[j]; y[j] := pom;
         j := j - 1;
      end;
   end;
end; { inicializuj }

var i, sucet : integer;
begin { hlavný program }
   inicializuj; { načítanie a utriedenie podľa y }
   for i := 2 to N do pocet[1, i] := 1;

   for i := 2 to N - 1 do spocitaj_z_bodu(i);

   sucet := 0;  { sčítaj hodnoty pocet[i,N] }
   for i := 1 to N - 1 do sucet := sucet + pocet[i, N];
   writeln(sucet);  { vypíš výsledok }
end.
```

P-III-3

Najskôr uvedieme jednoduchšie riešenie, ktoré pre každú permutáciu zostrojí sieť s $O(\log n)$ vrstvami a $O(n \log n)$ komparátormi. Neskôr ukážeme, ako toto riešenie zlepšiť tak, aby používalo iba $O(n)$ komparátorov. Obidve riešenia sa dajú implementovať ako algoritmy pracujúce v čase $O(n \log n)$.

Predpokladajme, že máme na vstupe permutáciu $A = a_1, a_2, \ldots a_n$ a nech $k = \lfloor n/2 \rfloor$ ($\lfloor x \rfloor$ označuje číslo x zaokrúhlené nadol na najbližšie celé číslo). Prvá vrstva našej siete bude mať tú vlastnosť, že po jej vykonaní budú na prvých k vodičoch čísla $1, 2, \ldots, k$ (nie nutne zotriedené) a na posledných $n-k$ vodičoch budú čísla $k+1, k+2, \ldots, n$ (nie nutne zotriedené). Táto vrstva sa zostaví takto. Nech S_1 je množina všetkých vodičov v hornej polovici, ktoré obsahujú čísla patriace do dolnej polovice (t.j. vodič i patrí do S_1, ak $i \leq k$ a $a_i > k$) a podobne nech S_2 je množina všetkých vodičov v dolnej polovici, ktoré obsahujú čísla z hornej polovice. Je zrejmé, že S_1 a S_2 majú rovnaký počet prvkov. V prvej vrstve každý vodič z S_1 spojíme komparátorom s jedným vodičom z S_2. V prípade, že na vstupe je naša permutácia A, každý takýto komparátor spôsobí výmenu a teda veľké hodnoty na komparátoroch v S_1 sa dostanú do dolnej polovice a

naopak. Po skončení prvej vrstvy teda platí požadovaná vlastnosť.

Po skončení prvej vrstvy rozdelíme n vodičov na dve skupiny: horných k a dolných $n-k$. Algoritmus rekurzívne zopakujeme na každej skupine zvlášť. Samozrejme, výsledné siete pre tieto dve skupiny vstupov môžu zdielať vrstvy, lebo pracujú na iných vodičoch. Rekurzia sa skončí vtedy, keď dostaneme skupinu obsahujúcu iba jeden vodič, lebo ten je automaticky utriedený. Keďže po každej vrstve sa nám veľkosť skupín zmenší zhruba na polovicu, celkový počet vrstiev je $O(\log n)$. V každej vrstve použijeme najviac $n/2$ komparátorov (viac ich v jednej vrstve ani byť nemôže), a preto celkový počet komparátorov je $O(n \log n)$.

Konštrukcia s $O(n)$ komparátormi funguje podobne, ale spája dvojice vodičov z S_1 a S_2 komparátormi šikovnejším spôsobom (nie ľubovoľne).

Najskôr zaveďme pojem *cyklus permutácie*. Permutáciu si môžeme predstaviť ako orientovaný graf, v ktorom vrcholy sú čísla $1, 2, \ldots, n$ a hrana vedie vždy z i do a_i. V takomto grafe z každého vrcholu práve jedna hrana vychádza a jedna do neho vchádza. Preto graf vždy pozostáva z niekoľkých cyklov. Napríklad permutácia $(7, 5, 4, 1, 2, 6, 8, 3)$ má 3 cykly: $(1 \to 7 \to 8 \to 3 \to 4 \to 1)$, $(2 \to 5 \to 2)$, $(6 \to 6)$.

Všimnime si, že našim cieľom je dostať číslo z vodiča i na vodič a_i. Čísla i a a_i sú ale v tom istom cykle, preto nie je potrebné vymieňať prvky navzájom medzi cyklami. Každý cyklus teda môžeme triediť osobitne. Takto sa nám úloha hneď na začiatku rozdelí na niekoľko menších podúloh. V najhoršom prípade však celá permutácia tvorí jeden cyklus, takže si nepomôžeme.

Teraz ukážeme, ako budeme spracovávať jeden konkrétny cyklus $(x_1 \to x_2 \to \cdots \to x_l \to x_1)$. Nech $k = \lfloor l/2 \rfloor$ je približne polovica dĺžky cyklu. Označme každé číslo v cykle znakom '+' alebo '−'. Znakom '−' označíme k najmenších čísel v cykle a znakom '+' označíme $n-k$ najväčších. Napríklad pre cyklus $(1 \to 7 \to 8 \to 3 \to 4 \to 1)$ označíme 1 a 3 znakom '−' a 4, 7 a 8 znakom '+'. Čísla označené mínusom budeme volať *malé* a čísla označené plusom budeme volať *veľké*.

Všimnime si súvislé úseky rovnakých znamienok v cykle (súvislý úsek môže pokračovať aj z x_l do x_1). Nech x_i je posledné číslo v niektorom súvislom úseku veľkých čísel (označených plusom). Nasledujúce číslo v cykle je teda malé, nasleduje teda úsek čísel označených mínusom. Nech x_j je posledné číslo v tomto úseku.

Na vodiči x_i je uložené číslo a_{x_i} (ďalšie číslo v cykle po x_i), ktoré je malé. Podobne, na vodiči x_j je uložené číslo a_{x_j}, ktoré je veľké. Číslo x_i je ale veľké a číslo x_j je malé, vodič x_i je teda položený nižšie, ako vodič x_j. Preto ak spojíme vodiče x_i a x_j komparátorom, musí určite dôjsť k výmene.

Pre každý súvislý úsek plusov v cykle takto nájdeme jeden komparátor. V našom príklade $(1 \to 7 \to 8 \to 3 \to 4 \to 1)$ čísla 7 a 8 tvoria prvý súvislý úsek plusov a 3 je nasledujúci úsek mínusov. Teda sieť bude obsahovať komparátor medzi vodičmi 8 a 3. Ďalší súvislý úsek plusov tvorí číslo 4 a nasledujúci úsek mínusov je 1. Teda pridáme komparátor medzi 1 a 4.

Popísaným postupom pre každý cyklus permutácie zostavíme komparátory prvej vrstvy. Ak je na vstupe táto permutácia, každý komparátor x_i, x_j spôsobí výmenu. Prvok a_{x_i} sa takto dostane na vodič x_j a vznikne samostatný cyklus obsahujúci iba pôvodný súvislý úsek mínusov. Všetky plusy z pôvodného cyklu budú po skončení v jednom novom cykle. Ak sme teda v niektorom cykle použili p komparátorov, cyklus sa rozpadne na $p+1$ nových cyklov. Každý komparátor teda pridá jeden cyklus. V ďalšej vrstve opäť nájdeme cykly v novej permutácii a pokračujeme rovnakým spôsobom. Skončíme vtedy, keď máme n jednoprvkových cyklov. Vtedy sú všetky čísla na svojich miestach.

Ak najväčší cyklus mal pred určitou vrstvou veľkosť x, tak po skončení tejto vrstvy bude mať veľkosť najviac $\lfloor x/2 \rfloor + 1$, lebo obsahuje buď iba plusy, alebo iba mínusy. Na začiatku máme cykly s veľkosťou najviac n a skončíme, keď všetky cykly majú veľkosť 1. Každou vrstvou veľkosť cyklov teda klesne približne na polovicu, z čoho vyplýva, že potrebujeme najviac $O(\log n)$

vrstiev. Každý komparátor zvýši počet cyklov o 1. Na začiatku máme aspoň jeden cyklus, na konci máme n cyklov. Použijeme teda najviac $n-1$ komparátorov.

Túto koštrukciu môžeme zapísať ako algoritmus, ktorý každú vrstvu vypíše v čase $O(n)$, celkový čas teda bude $O(n \log n)$. Pre každú vrstvu najprv v permutácii nájdeme cykly. To spravíme tak, že máme pomocné pole, v ktorom si pre každý prvok pamätáme číslo cyklu. Toto pole inicializujeme na nuly. Potom ideme postupne cez všetky prvky. Ak prvok i ešte nemá priradené číslo cyklu, znamená to, že sme objavili nový cyklus. Priradíme mu nové číslo a prejdeme po všetkých prvkoch tohto cyklu a označíme toto číslo do poľa. Po všetkých prvkoch prejdeme tak, že začneme v i a postupne ideme cez a_i, a_{a_i} atď., až kým neprídeme späť do i.

Keď máme označené všetky cykly, spočítame a uložíme si veľkosť každého cyklu a potom označíme čísla znamienkami '+' alebo '−'. Tu treba postupovať opatrne, aby sme zachovali lineárny čas. Pôjdeme znovu po číslach od 1 po n a pre každý cyklus si v poli pamätáme, koľko jeho členov sme už videli. Keď to číslo je viac ako polovica veľkosti cyklu, označujeme čísla '+', inak ich označujeme ako '−'. Nakoniec už len znova prejdeme cez cykly a keď nájdeme posledný prvok úseku plusov (t.j. i označené '+' také, že a_i je označené '−'), tak nájdeme koniec úseku mínusov začínajúcich prvkom a_i a pridáme komparátor. Potrebujeme si tiež poznačiť, ako sa zmení permutácia, čo budeme potrebovať pre konštrukciu novej vrstvy.

```
program P_III_3;
const MAX = 1000;
var a, nove_a, cyklus, velkost, najdene : array[1..MAX] of integer;
    cast : array[1..MAX] of char;
    n : integer;
    i, j, pocet_cyklov, velkost_cyklu : integer;
    hotovo : boolean;
begin
  {načítaj dáta}
  readln(n);
  for i := 1 to n do read(a[i]);

  {vypisuj jednotlivé vrstvy}
  hotovo := false;
  while not hotovo do begin { pre každú vrstvu }

    {nájdi cykly a ich veľkosti}
    for i := 1 to n do cyklus[i] := 0;
    pocet_cyklov := 0;
    for i := 1 to n do begin
      if cyklus[i] = 0 then begin {nový cyklus}
        j := i;
        inc(pocet_cyklov);
        velkost_cyklu := 0;
        while cyklus[j] = 0 do begin
          cyklus[j] := pocet_cyklov;
          j := a[j];
          inc(velkost_cyklu);
        end; {while}
        velkost[pocet_cyklov] := velkost_cyklu;
      end; {if}
```

```
      end; {for i}

      {označ prvky cyklov ako + a -}
      for i := 1 to pocet_cyklov do najdene[i] := 0;
      for i := 1 to n do begin
         inc(najdene[cyklus[i]]);
         if najdene[cyklus[i]] <= velkost[cyklus[i]] div 2 then
            cast[i] := '-'
         else
            cast[i] := '+';
      end; {for i}

      { skončili sme, ak máme n cyklov }
      if pocet_cyklov = n then hotovo := true
      else begin
         { nájdi komparátory a zapíš nové hodnoty poľa a }
         nove_a := a;
         for i := 1 to n do begin
            if (cast[i] = '+') and (cast[a[i]] = '-') then begin
               j := a[i]; {nájdi koniec úseku mínusov}
               while cast[a[j]] = '-' do j := a[j];
               writeln('Komparator (', i, ',', j, ')');
               nove_a[i] := a[j]; {výmena spôsobená komparátorom}
               nove_a[j] := a[i];
            end;
         end;
         a := nove_a;
         writeln('Koniec vrstvy');
      end; {else}
   end; {while}
end.
```

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

51. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia 3. kola kategórie P

1. súťažný deň

Vydala IUVENTA
pre vnútornú potrebu Ministerstva školstva SR
Autori príkladov B. Brejová, M. Forišek, M. Pál a T. Vinař
Sadzba programom LaTeX
Zodpovedný redaktor M. Forišek

MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

51. ročník, školský rok 2001/2002

Riešenia úloh 3. kola kategórie P

2. súťažný deň

P-III-4

Úlohu najprv preformulujeme do reči teórie grafov. Uzly budeme nazývať *vrcholmi*, rúry *hranami* a celú sústavu potrubí budeme volať *graf*. Postupnosť (nie nutne rôznych) vrcholov a hrán $v_0, e_1, v_1, e_2, \ldots, v_{k-1}e_k, v_k$ nazveme *sled*, ak každá hrana e_i spája vrcholy v_{i-1} a v_i. Sled nazveme *ťah*, ak sa v ňom žiadna hrana neopakuje.

Úlohou bolo v danom grafe G nájsť najmenšiu množinu ťahov takú, že žiadne dva nemajú spoločnú hranu a zjednotenie ich hrán obsahuje všetky hrany daného grafu. Alebo jednoduchšie povedané – keď si predstavíme graf ako obrázok, úlohou bolo nakresliť tento obrázok najmenším možným počtom ťahov.

Ako na to? Na úvod si uvedomme zopár jednoduchých skutočností. Keď máme nesúvislý graf, najlepšie riešenie nájdeme tak, že nájdeme najlepšie riešenie pre každý z jeho komponentov. Preto sa ďalej budeme zaoberať len súvislými grafmi. Súčet stupňov vrcholov grafu je párny, lebo je to dvakrát počet jeho hrán. Preto graf má nutne párny počet vrcholov nepárneho stupňa. Nech je ich $2k$, označme ich v_1, v_2, …, v_{2k}.

V každom z týchto vrcholov musí niektorý (aspoň jeden) ťah začínať alebo končiť. Prečo? Zoberme graf, v ktorom sú všetky vrcholy pôvodného grafu a hrany sú práve hrany niektorého ťahu. Všimnime si vrchol, v ktorom tento ťah nezačína ani nekončí. V tomto grafe má tento vrchol nutne párny stupeň – vždy, keď doň vôjdeme, hneď z neho aj vyjdeme, čím sa nám stupeň zväčší o 2. Stupeň konkrétneho vrcholu pôvodného grafu je zjavne súčtom jeho stupňov v grafoch zodpovedajúcich ťahom. Keď má teda tento stupeň byť nepárny, musí mať ten vrchol nepárny stupeň v aspoň 1 ťahu. To sa ale dá iba tak, že v ňom ten ťah začína alebo končí.

Máme teda $2k$ vrcholov a vieme, že v každom z nich začína alebo končí aspoň 1 ťah. Odtiaľ je zjavné, že potrebujeme aspoň k ťahov. Teraz ukážeme, že k ťahov nám stačí. (Okrem špeciálneho prípadu $k = 0$, kedy potrebujeme 1 ťah.)

Najskôr vyriešime jednoduchšiu úlohu. Majme súvislý graf, v ktorom sú všetky stupne vrcholov párne. Potom tento graf vieme nakresliť jedným uzavretým ťahom. Prečo? Zoberme najdlhší ťah. Tento je nutne uzavretý, lebo keby nebol, má v ňom posledný vrchol nepárny stupeň, a teda z neho vychádza nepoužitá hrana, ktorou by sme ho mohli predĺžiť. Keďže je uzavretý je jedno, v ktorom vrchole ho začneme. Keby neobsahoval všetky hrany, existuje vrchol v v tomto ťahu, z ktorého vychádza nepoužitá hrana. Potom ale vieme zostrojiť dlhší ťah tak, že začneme a skončíme náš pôvodný ťah vo vrchole v a navyše pridáme túto doteraz nepoužitú hranu, čo je spor. Ukázali sme teda, že najdlhší ťah v tomto grafe je uzavretý a obsahuje všetky hrany, čo je práve to, čo sme chceli. Uzavretý ťah, ktorý obsahuje všetky hrany grafu nazývame *Eulerovský*.

Späť k nášmu problému: ako nakresliť náš súvislý graf k ťahmi? Pospájajme ľubovoľne po dvojiciach vrcholy nepárneho stupňa. Na to použijeme práve k hrán a dostaneme tak súvislý graf, v ktorom majú všetky vrcholy párny stupeň. Ten vieme nakresliť jedným uzavretým ťahom. A keď z tohoto ťahu pridaných k hrán opäť vynecháme, rozpadne sa nám na hľadaných k ťahov. (Resp. pre $k = 0$ nám zostane 1 ťah.)

Ostáva jediný problém – ako efektívne nájsť Eulerovský ťah v danom súvislom grafe s párnymi stupňami vrcholov? Najjednoduchší na pochopenie je nasledujúci postup: nájdeme nejaký ťah a ten postupne predlžujeme, až kým neobsahuje všetky hrany. V našom programe je tento algoritmus implementovaný v rekurzívnej procedúre `vytvor_tah`. Táto procedúra začne v zadanom vrchole v a pridáva ďalšie a ďalšie hrany na koniec ťahu, až kým príde do vrcholu, z ktorého už nevedie nepoužitá hrana. Tento vrchol musí nutne byť v, lebo inak by musel mať nepárny stupeň (ťah doteraz použil párny počet hrán pre všetky vrcholy okrem v). Zostavili sme teda uzavretý ťah. Niektoré z vrcholov na tomto ťahu však ešte môžu mať nepoužité hrany. Pre každý taký vrchol u, ktorý nájdeme, zavoláme procedúru `vytvor_tah` rekurzívne. Tá nájde nový uzavretý ťah prechádzajúci cez vrchol u tak, aby už žiaden vrchol na tomto novom ťahu nemal nepoužité hrany. Nový ťah potom vložíme do pôvodného ťahu na tom mieste, kde pôvodný ťah prechádza cez u.

V programe potrebujeme vedieť pre daný vrchol rýchlo povedať, či má ešte nepoužitú hranu, a ak áno, tak ktorú. V programe máme pre každý vrchol zoznam jeho hrán a ešte špeciálnu premennú, ktorá ukazuje na prvú hranu v zozname, o ktorej nevieme, či je použitá, alebo nie. Vždy keď potrebujeme nepoužitú hranu, začneme z tohot miesta v zozname a ideme, až kým nenájdeme prvú skutočne nepoužitú hranu. Toto miesto si potom zaznačíme v tejto premennej pre ďalšie použitie. Počas celého programu teda prejdeme po zozname hrán pre vrchol iba raz a teda na hľadanie nepoužitých hrán minieme v celom programe čas $O(m)$.

Jednotlivé ťahy sú uložené ako spájané zoznamy hrán, aby sme mohli rýchlo vkladať rekurzívne nájdené uzavreté ťahy. Procedúra `vytvor_tah` pracuje v lineárnom čase v závislosti od počtu hrán komponentu. Najprv nájde počiatočný ťah a potom už len skontroluje všetky vrcholy na tomto ťahu a kde treba, zavolá sa rekurzívne. Vrcholy pridané rekurziou už netreba znovu kontrolovať. Celková časová aj pamäťová zložitosť je teda $O(m)$.

Kvôli zjednodušeniu implementácie program nespája nutne iba vrcholy nepárneho stupňa z toho istého komponentu, to však nemá vplyv na počet nájdených ťahov.

```
program P_III_4;
const MAXN = 201; MAXM = MAXN * MAXN + MAXN;
type
  hrana = record { typ: hrana }
            zac, kon : integer; { začiatočný a koncový vrchol }
            pridana : boolean; { je to jedna z umelo pridaných hrán? }
            pouzita : boolean; { je hrana použitá v ťahu? }
            dalsia : integer; { ďalšia hrana v ťahu }
  end;
  zoznam_sm = ^ zoznam; { typ: smerník na prvok zoznamu hrán }
  zoznam = record { typ: prvok zoznamu hrán }
             hrana : integer; { číslo hrany }
             dalsia : zoznam_sm; { smerník na ďalší prvok zoznamu }
  end;
var hrany : array[1..MAXM] of hrana; { pole hrán }
  susedia : array[1..MAXN] of zoznam_sm; { zoznam hrán pre každý vrchol }
  stupen : array[1..MAXN] of integer; { stupeň vrchola }
  nepouzite : array[1..MAXN] of zoznam_sm;
                { prvá hrana v zozname, ktorá môže byť nepoužitá }
  zaciatky : array[1..MAXN] of integer; { začiatky ťahov }
  pocet_zac : integer; { počet začiatkov v poli zaciatky }
  n, m : integer; { počet hrán a vrcholov }
```

```
procedure pridaj_suseda(komu, hrana : integer);
var zvysok : zoznam_sm;
begin { pridaj hranu "hrana" do zoznamu pre vrchol "komu" }
   zvysok := susedia[komu]; { odlož si zvyšok zoznamu }
   new(susedia[komu]); { vytvor nový prvok }
   susedia[komu]^.hrana := hrana;
   susedia[komu]^.dalsia := zvysok;
   inc(stupen[komu]); { zvýš stupeň vrcholu }
end; { pridaj_suseda }

procedure pridaj_hranu(index, zac, kon : integer; pridana : boolean);
begin { pridaj novú hranu na pozíciu "index" }
   hrany[index].zac := zac; hrany[index].kon := kon;
   hrany[index].dalsia := 0;
   hrany[index].pouzita := false;
   hrany[index].pridana := pridana;
   pridaj_suseda(zac, index); { pridaj do zoznamov pre oba koncové vrcholy }
   pridaj_suseda(kon, index);
end; { pridaj_hranu }

procedure nacitaj;
var i, zac, kon : integer;
begin { načítaj vstup a inicializuj polia }
   readln(n, m);
   for i := 1 to n do begin
      susedia[i] := nil;
      stupen[i] := 0;
   end;
   for i := 1 to m do begin
      readln(zac, kon);
      pridaj_hranu(i, zac, kon, false);
   end;
end; { nacitaj }

function nepouzita_hrana(vrchol : integer) : integer;
{ Vráti prvú nepoužitú hranu z vrcholu alebo 0, ak neexistuje.
  V "nepouzite[vrchol]" si uložíme hrany, ktoré už sme prehľadali
  (kvôli zrýchleniu ďalšieho prehľadávania). }
var p : zoznam_sm;
   nasiel : boolean;
begin
   p := nepouzite[vrchol];
   nasiel := false;
   while (p <> nil) and not nasiel do begin { preskoč použité hrany }
      if not hrany[p^.hrana].pouzita then nasiel := true
      else p := p^.dalsia;
   end;
   nepouzite[vrchol] := p; { ulož si preskočené hrany }
```

```
    { našli sme nepoužitú hranu ? }
    if nasiel then nepouzita_hrana := p^.hrana
    else nepouzita_hrana := 0;
end; { nepouzita_hrana }

function druhy_koniec(hrana, koniec : integer) : integer;
begin { vráť ten koniec hrany, ktorý nie je rovný "koniec". }
    if hrany[hrana].zac = koniec then
        druhy_koniec := hrany[hrana].kon
    else
        druhy_koniec := hrany[hrana].zac;
end; { druhy_koniec }

procedure zorientuj_hranu(hrana, koniec : integer);
begin { otoč hranu tak, aby mala koniec v "koniec" }
    if hrany[hrana].zac = koniec then begin
        hrany[hrana].zac := hrany[hrana].kon;
        hrany[hrana].kon := koniec;
    end;
end;

procedure vytvor_tah(vrchol : integer;
                     var prva_hrana, posledna_hrana : integer;
                     var bola_pridana : boolean);
{ Rekurzívne vytvor uzavretý ťah začínajúci vo "vrchol" taký,
  že každý vrchol na ňom má použité všetky hrany. Vráť prvú
  a poslednú hranu a tiež či sme pridali aspoň jednu pridanú hranu. }
var akt, hrana, dalsia_hrana, nova_prva, nova_posledna : integer;
    nova_bola : boolean;
begin
    bola_pridana := false;
    { nájdi cyklus začínajúci vo vrchole "vrchol" }
    akt := vrchol;   { aktuálny vrchol }
    hrana := nepouzita_hrana(akt);
    prva_hrana := hrana;
    while hrana <> 0 do begin        { kým vieme cyklus predĺžiť, predĺžme }
        posledna_hrana := hrana;
        hrany[hrana].pouzita := true;
        akt := druhy_koniec(hrana, akt);
        hrana := nepouzita_hrana(akt);
        hrany[posledna_hrana].dalsia := hrana;
        zorientuj_hranu(posledna_hrana, akt);
    end;

    { prejdi cyklus znova a doplň ďalšie cykly, kde chýbajú }
    hrana := prva_hrana;
    akt := vrchol;
    while hrana <> 0 do begin
        if hrany[hrana].pridana then bola_pridana := true;
```

```
        dalsia_hrana := hrany[hrana].dalsia;
        if nepouzita_hrana(akt) <> 0 then begin { ešte je čo pridať }
            vytvor_tah(akt, nova_prva, nova_posledna, nova_bola);
            if nova_bola then bola_pridana := true;
            hrany[hrana].dalsia := nova_prva;
            hrany[nova_posledna].dalsia := dalsia_hrana;
        end;
        akt := druhy_koniec(hrana, akt);
        hrana := dalsia_hrana;
    end;

    { zacyklíme nájdený cyklus }
    hrany[posledna_hrana].dalsia := prva_hrana;
end;

procedure pridaj_hrany;
var i, posledny : integer;
begin { pridaj hrany medyi vrcholmi nepárneho stupňa }
    pocet_zac := 0;
    posledny := 0; { číslo posledného nepárneho nespárovaného vrcholu }
    for i := 1 to n do begin
        if stupen[i] mod 2 = 1 then begin
            if posledny = 0 then posledny := i
            else begin
                inc(m);
                pridaj_hranu(m, posledny, i, true);
                inc(pocet_zac);
                zaciatky[pocet_zac] := m;
                posledny := 0;
            end; {else}
        end; {if stupen}
    end; {for i}
end;

procedure vypis_tah(zmazana_hrana : integer);
var hrana : integer;
begin { vypíš tah pokračujúci za hranou "zmazana_hrana" }
    hrana := hrany[zmazana_hrana].dalsia; { prvá hrana, ktorú vypíšeme }
    while (hrana <> zmazana_hrana) and (not hrany[hrana].pridana) do begin
        write(hrany[hrana].zac, ' ');
        hrana := hrany[hrana].dalsia;
    end;
    write(hrany[hrana].zac); { vypíš druhý koniec posledne vypísanej hrany }
    if not hrany[hrana].pridana then { ak sme v komponente bez pridanej hrany }
        writeln(' ', hrany[hrana].kon) { vypíšeme ešte jeden vrchol }
    else
        writeln;
end;
```

```
var i, prva_hrana, posledna_hrana : integer;
    bola_pridana : boolean;
begin { Hlavný program }
   nacitaj;   { vstup a inicializácia }
   pridaj_hrany;   { pridanie hrán, aby stupňe boli párne }
   for i := 1 to n do nepouzite[i] := susedia[i]; { inicializuj "nepouzite" }
   { pre každý vrchol, ktorý má neprejdené hrany prejdi komponent }
   for i := 1 to n do begin
      if nepouzita_hrana(i) <> 0 then begin
         vytvor_tah(i, prva_hrana, posledna_hrana, bola_pridana);
         if not bola_pridana then begin { párny komponent pridaj do začiatkov }
            inc(pocet_zac);
            zaciatky[pocet_zac] := prva_hrana;
         end;
      end;
   end;
   { výpis výsledku }
   writeln(pocet_zac);
   for i := 1 to pocet_zac do begin
      vypis_tah(zaciatky[i]);
   end;
end.
```

P-III-5

Idea riešenia je veľmi jednoduchá. Postupne po rade generujeme všetky pozície s najviac N kameňmi, z ktorých sa dá dosiahnuť cieľová pozícia. Všetky nájdené pozície si pamätáme, a pre každú novovygenerovanú pozíciu skontrolujeme, či sme ju už niekedy vygenerovali. Ak nie, zapamätáme si ju a zvýšime počítadlo nájdených pozícií o jedna.

Pozície generujeme odzadu. Začneme z cieľovej pozície a vygenerujeme všetky možnosti pre posledný ťah. Posledný ťah musel byť skok doľava alebo skok doprava. Ak posledným ťahom bol skok doprava z pozície i, po tomto skoku musia byť pozície i a $i+1$ prázdne a pozícia $i+2$ musí obsahovať kameň. Tesne pred týmto skokom museli pozície i a $i+1$ obsahovať kameň a pozícia $i+2$ musela byť prázdna. Ak teda chceme vygenerovať všetky možné pozície z ktorých sa dá dostať do pozície p skokom doprava, jednoducho pre každú trojicu po sebe idúcich políčok pozície p obsahujúcu podpostupnosť 001, nahradíme túto podpostupnosť trojicou 110. Podobne pre skok doľava hľadáme všetky výskyty trojice 100 a nahrádzame ju trojicou 011.

Pre každú uvažovanú pozíciu p vieme teda vygenerovať všetky pozície, z ktorých sa do p dá dostať na jeden ťah. Procedúra `generuj` rekurzívne generuje všetky pozície, z ktorých sa dá dostať do danej pozície. Pre danú pozíciu p vygeneruje všetkých jej priamych (na jeden ťah) predchodcov. Pre každého q predchodcu si overí, či už bol vygenerovaný a ak nie, rekurzívne sa zavolá na q.

Na to, aby sme vedeli zisťovať, či sme už danú pozíciu vygenerovali alebo nie, potrebujeme dátovú štruktúru, ktorá dovoľuje pridávať nové pozície a rýchlo zisťovať, či už bola daná pozícia niekedy pridaná. Dátových štruktúr podporujúcich tieto operácie je množstvo, napríklad hašovacie tabuľky alebo rôzne varianty vyhľadávacích stromov. My sme sa pre jednoduchosť implementácie rozhodli použiť tzv. písmenkový strom (po anglicky trie).

Písmenkový strom je dátová štruktúra na uchovávanie reťazcov núl a jednotiek. Keďže každá pozícia sa dá reprezentovať ako postupnosť núl a jednotiek, písmenkové stromy nám prídu

celkom vhod. Písmenkový strom je strom, v ktorom každý vrchol má najviac dvoch potomkov. Každá cesta od koreňa stromu k vrcholu zodpovedá reťazcu – ak i-ta hrana tejto cesty vedie od vrchola k jeho ľavému potomkovi, i-ty znak reťazca je 0, ak k pravému, i-ty znak je 1. Pre každý vrchol si pamätáme, či reťazec zodpovedajúci ceste do tohto vrchola bol skutočne vložený do stromu (mohlo by sa stať že daný vrchol bol vytvorený len preto, že leží na ceste zodpovedajúcej nejakému dlhšiemu reťazcu). Okrem toho si pre každý vrchol pamätáme ukazovateľ na jeho ľavého a pravého potomka (ak daného potomka má). Zistenie, či daný reťazec bol vložený do stromu je veľmi jednoduché – ideme po ceste z koreňa de novanej týmto reťazcom. Ak sa nám podarí dôjsť až na koniec a vrchol, v ktorom skončíme, je označený ako vrchol zodpovedajúci vloženému reťazcu, tento reťazec bol niekedy vložený, inak nie. Vkladanie reťazca je rovnako jednoduché – sledujeme cestu zodpovedajúcu reťazcu a ak narazíme na miesto, kde nemôžeme ísť ďalej lebo ďalšie vrcholy cesty neexistujú, jednoducho potrebné vrcholy vytvoríme.

Časová a pamäťová zložitosť nášho programu závisí od výsledného počtu pozícií P. Pre každú nájdenú pozíciu potrebujeme vytvoriť najviac K uzlov v strome a preto pamäť je $O(PK)$. Pre každú nájdenú pozíciu musíme prezrieť všetkých $2K-4$ potenciálnych ťahov a každý možný ťah ešte skúšame vložiť do stromu v čase $O(K)$. Teda celkový čas je $O(PK^2)$.

```
program P_III_5;
const MAX = 200;
type uzol_sm = ^ uzol;     { typ: smerník na uzol stromu }
     uzol = record         { typ: uzol stromu }
       vetva : array[boolean] of uzol_sm; { smerníky na deti }
       vlozeny : boolean;      { či bol zodpovedajúci reťazec vložený }
     end;
     pozicia = array[1..MAX] of boolean; { typ: pozícia hry }
var koren : uzol_sm;           { koreň písmenkového stromu }
    K, N : integer;            { K a N zo vstupu }
    pocet_pozicii : integer;   { počet objavených pozícií }

function novy_uzol : uzol_sm;  { vytvor nový uzol stromu }
var p : uzol_sm;
begin
  new(p);                  { alokácia pamäte }
  p^.vetva[false] := nil;  { uzol nemá žiadne deti }
  p^.vetva[true] := nil;
  p^.vlozeny := false;     { zodpovedajúci reťazec nebol vložený }
  novy_uzol := p;          { vráť vytvorený uzol }
end; { novy_uzol }

function pridaj(var poz : pozicia) : boolean;
{ pridaj pozíciu "poz" do stromu, vráť true, ak tam ešte nebola }
var i : integer;
    p : uzol_sm;  { aktuálny uzol stromu }
begin
  p := koren;    { začneme v koreni }
  for i := 1 to K do begin
    if p^.vetva[poz[i]] = nil then { ak vetva neexistuje, vytvoríme }
      p^.vetva[poz[i]] := novy_uzol;
    p := p^.vetva[poz[i]];        { posuňme sa po príslušnej vetve }
```

```
    end;
    if p^.vlozeny then pridaj := false { pozícia už je v strome }
    else begin { pozícia nie je v strome }
        p^.vlozeny := true;     { pridáme ju }
        inc(pocet_pozicii);     { zvýšime počet objavených pozícií }
        pridaj := true;
    end;
end; { pridaj }

procedure prirad(var poz : pozicia; kde : integer; b1, b2, b3 : boolean);
{ pomocná procedúra, ktorá priradí hodnoty trom políčkam pozície }
begin
    poz[kde] := b1; poz[kde+1] := b2; poz[kde+2] := b3;
end; { prirad }

procedure generuj(var poz : pozicia; pocet_kamenov : integer);
{ rekurzívne generuj všetky pozície z pozície "poz",
"pocet_kamenov" je počet kamenňov v "poz". }
var i : integer;
begin
    if pocet_kamenov < N then begin
        for i := 1 to K - 2 do begin
            { skús ťah z pozície i+2 doľava }
            if poz[i] and (not poz[i+1]) and (not poz[i+2]) then begin
                prirad(poz, i, false, true, true); { zmeň "poz" }
                if pridaj(poz) then generuj(poz, pocet_kamenov+1);
                prirad(poz, i, true, false, false); { obnov "poz" }
            end;
            { skús ťah z pozície i doprava }
            if (not poz[i]) and (not poz[i+1]) and poz[i+2] then begin
                prirad(poz, i, true, true, false); { zmeň "poz" }
                if pridaj(poz) then generuj(poz, pocet_kamenov+1);
                prirad(poz, i, false, false, true); { obnov "poz" }
            end;
        end;
    end;
end; { generuj }

var i, kamen, pocet_kamenov : integer;
    poz : pozicia;
begin { hlavný program }
    { načítaj a spočítaj kamene na vstupe }
    readln(K, N);
    pocet_kamenov := 0;
    for i := 1 to K do begin
        read(kamen);
        poz[i] := (kamen = 1);
        if poz[i] then inc(pocet_kamenov);
    end;
```

```
    { inicializuj globálne premenné }
    koren := novy_uzol;
    pocet_pozicii := 0;
    { generuj z danej pozície }
    if pocet_kamenov <= N then begin
      pridaj(poz);
      generuj(poz, pocet_kamenov);
    end;
    writeln(pocet_pozicii); { vypíš výsledok }
end.
```

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY
51. ROČNÍK MATEMATICKEJ OLYMPIÁDY
Riešenia 3. kola kategórie P
2. súťažný deň

Vydala IUVENTA
pre vnútornú potrebu Ministerstva školstva SR
Autori príkladov B. Brejová, M. Forišek, M. Pál a T. Vinař
Sadzba programom LaTeX
Zodpovedný redaktor M. Forišek