Slovenská komisia Matematickej olympiády
FMFI UK, Mlynská dolina, 842 48 Bratislava 4

50. ročník Matematickej olympiády, školský rok 2000/2001

Riešenia úloh 2. kola kategórie P

P-II-1 *(Trojuholníky)*

Predpokladajme, že máme dĺžky drevených paličiek zotriedené podľa veľkosti, t.j. $d_1 < d_2 < \ldots < d_N$. Potom trojica indexov $i < j < k$ určuje paličky tvoriace trojuholník práve vtedy, keď $d_k < d_i + d_j$. Zostávajúce dve trojuholníkové nerovnosti sú totiž splneneé triviálne, lebo $d_i, d_j < d_k$. Pre každú dvojicu indexov $i < j$ označme symbolom $l(i, j)$ najväčšie číslo k také, že $d_k < d_i + d_j$; všimnime si, že $l(i, j) \geq j$. Zvolená dvojica indexov $i < j$ sa dá na trojicu $i < j < k$ určujúcu trojuholník doplniť práve tými k, pre ktoré platí $j < k \leq l(i, j)$. Pre pevnú dvojicu indexov i a j teda existuje práve $l(i, j) - j$ takých k, že paličky s indexami $i < j < k$ tvoria trojuholník. Na určenie počtu trojuholníkov teda stačí určiť hodnoty $l(i, j)$ pre všetky $i < j$ a sčítať výrazy $l(i, j) - j$.

Z de nície $l(i, j)$ vyplýva, že $N = l(i, N) \geq l(i, N - 1) \geq l(i, N - 2) \geq \ldots \geq l(i, i + 1)$. Náš program bude pracovať nasledovne: Pre každé i spočítame hodnoty $l(i, N - 1), l(i, N - 2), \ldots, l(i, i + 1)$ a súčasne budeme počítať súčet $(l(i, N - 1) - (N - 1)) + \cdots + (l(i, i + 1) - (i + 1))$, ktorý predstavuje počet trojuholníkov, ktorých najkratšia strana má dĺžku d_i. Hodnotu $l(i, j)$ spočítame tak, že hodnotu $l(i, j + 1)$ budeme zmenšovať o jedničku tak dlho, kým $d_{l(i,j)} \geq d_i + d_j$. Pretože celkový počet zmenšení o jedničku počas výpočtu hodnôt $l(i, N - 1), l(i, N - 2), \ldots, l(i, i + 1)$ je najviac $N - 2$, je čas výpočtu pre jedno pevné i lineárny od N. Celkový čas výpočtu pre všetkých $N - 2$ možných hodnôt i je teda $O(N^2)$. V tomto čase môžeme ľahko spraviť aj úvodné triedenie dĺžok paličiek. Časová zložitosť nášho algoritmu je teda $O(N^2)$ a pamäťová $O(N)$.

```
program trojuhelniky;            { P-II-1 }
const MAXN=1000;
var N:word;                      { počet tyček }
    T:longint;                   { počet trojúhelníků }
    i,j,k:word;
    d:array[1..MAXN] of real;    { délky tyček }
    e:real;
begin
 read(N);
 for i:=1 to N do read(d[i]);
 for i:=1 to N-1 do          { setřídíme délky tyček }
  for j:=i+1 to N do
   if d[i]>d[j] then
    begin
     e:=d[i]; d[i]:=d[j]; d[j]:=e
```

```
    end;
 T:=0;
 for i:=1 to N-2 do          { i - nejkratší tyčka z trojice }
  begin
   j:=N-1; k:=N;             { j - druhá nejkratší tyčka z trojice }
   repeat
    while (j<k) and (d[k]>=d[i]+d[j]) do
                             { podmínku (j<k) lze vypustit }
     dec(k);                 { hledáme nejdelší tyčku do trojice }
    T:=T+k-j;
    dec(j);
   until j=i;
  end;
 writeln(T);
end.
```

P-II-2 *(Robot)*

Najskôr si rozmyslime, ako by sme zadanú úlohu riešili, keby sme chceli nájsť najkratšiu cestu robota medzi dvoma zadanými políčkami. Algoritmus na riešenie tejto úlohy je známy pod názvom prehľadávanie do šírky alebo tiež algoritmus vlny. Každému políčku v priebehu výpočtu priradíme číslo, ktoré udáva minimálny počet krokov, ktoré robot potrebuje na premiestnenie sa zo začiatočného políčka na uvažované políčko. Algoritmus pracuje vo fázach. Najskôr začiatočnému políčku priradí nulu. V i-tej fáze priradí číslo i všetkým políčkam, ktoré susedia s nejakým políčkom, ktorému bolo priradené číslo $i-1$ a ktorým sme doteraz žiadne číslo nepriradili. Je zrejmé, že takto priradené čísla určujú minimálny počet krokov potrebný na premiestnenie robota zo začiatočného políčka na každé z políčok.

Teraz si rozmyslime, ako sa tento algoritmus dá modi kovať tak, aby riešil úlohu zo zadania. V i-tej fáze nebudeme číslovať políčka vo vzdialenosti i krokov, ale políčka, na ktoré sa dá presunúť cestou s i zmenami smeru. To sú zjavne políčka, ktoré ešte nemajú číslo, ležia v rovnakom riadku alebo stĺpci ako niektoré políčko s číslom $i-1$ a nie sú od neho oddelené prekážkou. Správnosť tohto algoritmu je zrejmá. Zostáva domyslieť detaily jeho implementácie. Políčka si budeme ukladať v poli tak, že políčka s rovnakým číslom budú tvoriť súvislé úseky a políčka s nižšími číslami budú pred políčkami s vyššími číslami. Pokiaľ prídeme počas nejakej fázy na ešte neočíslované políčko, zaradíme ho na koniec poľa. V priebehu algoritmu vyberieme vždy prvé nespracované políčko z poľa a prehľadáme jeho riadok a stĺpec. Pole, s ktorým sa pracuje práve popísaným spôsobom, sa obvykle nazýva fronta (políčka sa stavajú na koniec fronty a čakajú, kým na ne príde rad). Nech M a N sú rozmery štvocovej siete, potom spracovanie každého z $M \times N$ políčok vyžaduje čas $O(M+N)$. Celková časová zložitosť nášho algoritmu by tedy bola $O(M^2N + MN^2)$.

Čas potrebný na výpočet sa však ešte dá zlepšiť. Jeden súvislý úsek riadku alebo stĺpca totiž prehľadávame niekoľkokrát – pre každé jeho políčko raz. Preto si budeme pre každé políčko pamätať, či sme už prehľadali súvislý úsek (bez

prekážok) riadku, resp. stĺpca, v ktorom toto políčko leží. Pred prehľadávaním sa najskôr pozrieme, či sme tento úsek riadku alebo stĺpca už neprehľadávali. Ak áno, už ho neprehľadávame. Ak nie, prezrieme ho a pre všetky jeho políčka si zapamätáme, že sme daný úsek už prehľadávali. Všimnite si, že pre každé políčko si musíme samostatne pamätať, či bol prehľadaný úsek v riadku a či bol prehľadaný úsek v stĺpci.

Čas potrebný na načítanie vstupných dát, na prácu s frontou a na výpis riešenia je zrejme $O(MN)$. Zostáva stanoviť čas potrebný na prehľadávanie riadkov a stĺpcov. Každý súvislý úsek bez prekážok prehľadáme práve raz, súčet ich dĺžok je $O(MN)$, lebo každé políčko je nanajvýš v dvoch (riadok a stĺpec). Súvislý úsek ľahko prehliadneme v čase lineárnom od jeho dĺžky, preto aj celková časová zložitosť nášho algoritmu je $O(MN)$; pamäťová zložitosť je tiež $O(MN)$.

```
program robot;          { P-II-2 }
const MAX=20;           { maximální rozměry čtvercové sítě }
type policko = record
                x,y : word;
               end;
var sirka, vyska: word; { rozměry místnosti }
    prekazky: array[1..MAX,1..MAX] of boolean;
                        { rozložení překážek v místnosti }
    navstiveno, svisle, vodorovne: array[1..MAX,1..MAX] of boolean;
                        { indikátory navštívení políček místnosti }
    predchozi: array[1..MAX,1..MAX] of policko;
                        { předchozí políčko na optimální cestě }
    start, cil: policko;
                        { počáteční a cílové políčko }
    fronta: array[1..MAX*MAX] of policko;
    zpracovano, vefronte: word;
                        { fronta prohledávání do šířky }
    x, y: word; { pomocné proměnné }
    i: integer;
procedure vypis( x, y: word);
 var x0, y0: word;
     k: integer;
 begin
  x0:=predchozi[x,y].x; { předchozí políčko na optimální cestě }
  y0:=predchozi[x,y].y;
  if ( x0 = predchozi[x0,y0].x ) and ( y0 = predchozi[x0,y0].y )
    then begin
      { Jsme na počátečním políčku ... }
      if x0 < x then write('Program pro robota '
                           +'(počáteční natočení DOLŮ):');
      if x0 > x then write('Program pro robota '
                           +'(počáteční natočení NAHORU):');
      if y0 < y then write('Program pro robota '
```

```
                          +'(počáteční natočení DOPRAVA):');
      if y0 > y then write('Program pro robota '
                          +'(počáteční natočení DOLEVA):');
    end
  else
    begin
      { Nejprve vypíšeme předchozí políčko
        a potom směr našeho otočení }
      vypis(x0,y0);
      k:=(x-x0)*(y0-predchozi[x0,y0].y)-
         (x0-predchozi[x0,y0].x)*(y-y0);
      if k > 0 then write(' <DOPRAVA>');
      if k < 0 then write(' <DOLEVA>');
    end;
  k:=x+y-x0-y0; { Spočítáme počet kroků, které je třeba udělat }
  if k < 0 then k:=-k;
  while k > 0 do
    begin
      write(' <KROK>');
      dec(k)
    end
 end;
begin
  { Načteme rozměry sítě, překážky, počáteční a cílové políčko }
  readln(vyska,sirka);
  readln(start.x,start.y);
  readln(cil.x,cil.y);
  for x:= 1 to vyska do
    for y:= 1 to sirka do
      begin
        read(i);
        prekazky[x,y]:=(i=1);
        navstiveno[x,y]:=false;
        svisle[x,y]:=false;
        vodorovne[x,y]:=false;
      end;
  { Test na shodu počátečního a cílového políčka }
  if ( start.x = cil.x ) and ( start.y = cil.y ) then
    begin
      writeln('Počáteční a cílové políčko jsou stejné.');
      halt;
    end;
  { Inicializace prohledávání do šířky }
  zpracovano:=0;
  vefronte:=1;
  fronta[1]:=start;
```

```
navstiveno[start.x,start.y]:=true;
predchozi[start.x,start.y]:=start;
{ Cyklus prohledávání do šířky }
while ( zpracovano < vefronte ) do
  begin
    inc(zpracovano);
    x:=fronta[zpracovano].x;
    y:=fronta[zpracovano].y;
    if not vodorovne[x,y] then
      begin
        { Projedeme síť vodorovně ( řádek ) }
        vodorovne[x,y]:=true;
        i:=1;
        while ( y+i <= sirka ) and not ( prekazky[x,y+i] ) do
          begin
            vodorovne[x,y+i]:=true;
            if not navstiveno[x,y+i] then
              begin
                navstiveno[x,y+i]:=true;
                predchozi[x,y+i]:=fronta[zpracovano];
                inc(vefronte);
                fronta[vefronte].x:=x;
                fronta[vefronte].y:=y+i;
              end;
            inc(i);
          end;
        i:=-1;
        while ( y+i >= 1 ) and not ( prekazky[x,y+i] ) do
          begin
            vodorovne[x,y+i]:=true;
            if not navstiveno[x,y+i] then
              begin
                navstiveno[x,y+i]:=true;
                predchozi[x,y+i]:=fronta[zpracovano];
                inc(vefronte);
                fronta[vefronte].x:=x;
                fronta[vefronte].y:=y+i;
              end;
            dec(i);
          end;
      end;
    if not svisle[x,y] then
      begin
        { Projedeme síť svisle ( sloupeček ) }
        svisle[x,y]:=true;
        i:=1;
```

```
          while ( x+i <= vyska ) and not ( prekazky[x+i,y] ) do
            begin
              svisle[x+i,y]:=true;
              if not navstiveno[x+i,y] then
                begin
                  navstiveno[x+i,y]:=true;
                  predchozi[x+i,y]:=fronta[zpracovano];
                  inc(vefronte);
                  fronta[vefronte].x:=x+i;
                  fronta[vefronte].y:=y;
                end;
              inc(i);
            end;
          i:=-1;
          while ( x+i >= 1 ) and not ( prekazky[x+i,y] ) do
            begin
              svisle[x+i,y]:=true;
              if not navstiveno[x+i,y] then
                begin
                  navstiveno[x+i,y]:=true;
                  predchozi[x+i,y]:=fronta[zpracovano];
                  inc(vefronte);
                  fronta[vefronte].x:=x+i;
                  fronta[vefronte].y:=y;
                end;
              dec(i);
            end;
        end;
    end;
  if not navstiveno[cil.x,cil.y] then
    begin
      { Na cílové políčko se nelze dostat ... }
      writeln('Cesta z počátečního na cílové políčko neexistuje.');
      halt
    end;
  { A vypíšeme nalezenou cestu ... }
  vypis(cil.x,cil.y);
  writeln;
end.
```

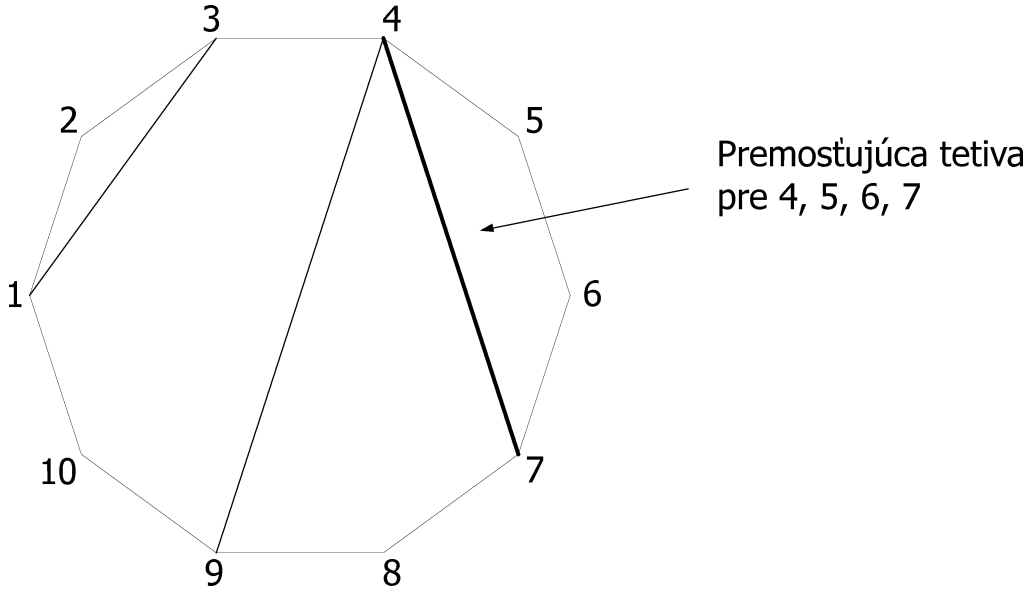
P-II-3 *(Jožko)*

Úlohu si najskôr trošku preformulujme. Obraz je vlastne konvexný n-uholník a strihy sú jeho nepretínajúce sa tetivy. Úlohou je nájsť mnohouholník tvorený stranami pôvodného mnohouholníka a tetivami, v ktorom neleží žiadna tetiva a ktorý má zo všetkých takýchto mnohouholníkov najviac vrcholov. Ďalej ho budeme označovať ako *najväčší mnohouholník*. Pokiaľ nebude povedané inak,

slovom mnohouholník rozumieme jeden z mnohouholníkov, na ktoré sa pôvodný mnohouholník rozpadol.

Najskôr si všetky tetivy otočme tak, aby začiatočný vrchol bol menší ako koncový a zotrieďme si ich. Pri triedení porovnávame najskôr číslo začiatočného vrchola, v prípade rovnosti sa berie obrátené poradie čísel koncových vrcholov (teda $(1,2) < (2,3)$, ale $(1,2) > (1,3)$). Po zotriedení už môžeme začať hľadať najväčší mnohouholník.

Nazvime *premosťujúcou tetivou* mnohouholníka tú spomedzi všetkých tetív a strán tohto mnohouholníka, ktorá má najmenšie číslo začiatočného vrchola a najväčšie číslo koncového vrchola. Zjavne mnohouholník s premosťujúcou tetivou (i, j) bude obsahovať vrcholy i, j a ešte nejaké vrcholy z $i \ldots j$. Tiež si všimnime, že každá z pôvodných tetív je premosťujúcou tetivou práve jedného mnohouholníka (a premosťujúcou tetivou mnohouholníka, ktorý obsahuje hranu $(1, n)$ je práve táto hrana). Hovoríme, že mnohouholník je svojou premosťujúcou tetivou *ohraničený*.



A teraz ako nájsť najväčší mnohouholník. Použijeme nasledovnú ideu: Algoritmus postupne prechádza vrcholy n-uholníka od 1 do n. Udržuje si pri tom zásobník, v ktorom sú uložené tetivy, u ktorých prešiel ich počiatočným vrcholom, ale ešte nie koncovým. Sú to teda *premosťujúce tetivy* pre doteraz neuzavreté mnohouholníky. Pre každú tetivu na zásobníku si ešte pamätáme doteraz narátaný počet vrcholov v ňou ohraničenom mnohouholníku. Keďže aktuálny vrchol vždy patrí do mnohouholníka ohraničeného najneskôr začínajúcou premosťujúcou tetivou, stačí vždy upravovať len počet vrcholov pre tetivu na vrchu zásobníka.

Konkrétna implementácia vyzerá nasledovne: Vždy, keď sa posunieme do ďalšieho vrchola, postupne odoberieme zo zásobníka tetivy, ktoré v tomto vrchole končia. Keďže tetivy sa nepretínajú, sú všetky tieto tetivy momentálne na vrchu zásobníka. Prešli sme totiž všetky vrcholy, ktoré mohli ležať v mnohouholníkoch ohraničených týmito tetivami. K týmto tetivám už teda boli spočítané počty vrcholov v nimi ohraničených mnohouholníkoch, a tak stačí podľa nich upraviť maximum. Po každom odobraní tetivy zo zásobníka prirátame jedna k počtu vr-

cholov pre tetivu na vrchu zásobníka, lebo do príslušného mnohouholníka patrí aj aktuálny vrchol. Potom pridáme všetky tetivy začínajúce v danom vrchole do zásobníka. Počty vrcholov u nových tetív sa nastavia na jedna, lebo sa musí zarátať aktuálny vrchol. Vďaka zotriedeniu tetív stačí tetivy len zaradom odoberať z poľa, kým sa ich začiatočný vrchol zhoduje s aktuálnym. Zotriedenie tak zaistí, že z tetív, ktoré začínajú v aktuálnom vrchole, budú tie, ktoré neskôr skončia, vložené do zásobníka skôr. Keď sú pridané všetky tetivy začínajúce v aktuálnom vrchole, priráta sa jedna k počtu vrcholov tetivy na vrchu zásobníka za vrchol, do ktorého sa presúvame.

Práve uvedený algoritmus sa dá ešte zrýchliť – stačí si uvedomiť, že je zbytočné posúvať sa po obvode len po jednom vrchole. Stačí nam vlastne len obísť vrcholy, v ktorých niektorá tetiva začína alebo končí. To, o koľko sa mám posunúť, ľahko zistím ako minimum z konca tetivy na vrchu zásobníka a začiatku prvej ešte nezaradenej tetivy. Získame tak algoritmus s časovou zložitosťou $O(k \log k)$. Samotný prechod n-uholníkom nám bude trvať len $O(k)$, pretože každú tetivu len raz vložíme na zásobník a raz ju odtiaľ vyberieme. Správnosť algoritmu bola ukázaná v popise.
Program je priamou implementáciou algoritmu:

```
program Pepik;  { P-II-3 }
const
  MAXK = 100;
type
  Cut = record
          a, b : Integer;
        end;
  CutA = Array[1..MAXK] of cut;
var
  n, k : Integer;       {Počet vrcholů; Počet střihů}
  c : CutA;             {Popis jednotlivých střihů}

{Načte vstup}
procedure ReadInp;
var
  i, tmp : Integer;
begin
  Write('Zadejte pocet vrcholu a pocet strihu: ');
  Read(n, k);
  Write('Zadejte jednotlive strihy: ');
  {Načte popis střihu}
  for i := 1 to k do begin
    Read(c[i].a, c[i].b);
    {První číslo bude menší}
    if c[i].a > c[i].b then begin
      tmp := c[i].a;
      c[i].a := c[i].b;
```

```
      c[i].b := tmp;
    end;
  end;
  {Přidáme pomocnou tětivu}
  Inc(k);
  c[k].a := 1;
  c[k].b := n;
end;

{Porovná dva střihy}
function CmpCut(a, b : cut) : ShortInt;
begin
  if (a.a < b.a) or ((a.a = b.a) and (a.b > b.b)) then
    CmpCut := -1
  else if (a.a = b.a) and (a.b = b.b) then
    CmpCut := 0
  else
    CmpCut := 1;
end;

{Setřídí pole se střihy QuickSortem}
procedure SortCut(d, u : Integer);
var
  m, tmp : cut;          {Pivot}
  i, j : Integer;
begin
  m := c[(d+u) div 2];  {Vybereme pivota}
  i := d; j := u;
  while i <= j do begin
    {Nalezneme prvky ve špatných částech}
    while CmpCut(c[i], m) = -1 do
      Inc(i);
    while CmpCut(c[j], m) = 1 do
      Dec(j);
    if i <= j then begin
      {Zaměníme prvky ve špatných částech}
      tmp := c[i];
      c[i] := c[j];
      c[j] := tmp;
      Inc(i);
      Dec(j);
    end;
  end;
  if i < u then  {Je co třídit v pravé části?}
    SortCut(i, u);
  if d < j then  {Je co třídit v levé části?}
```

```
    SortCut(d, j);
end;

{Nalezne největší mnohoúhelník}
function FindMax(n, d, u : Integer) : Integer;
var
  StackC : Array[1..MAXK] of Integer;
  StackN : Array[1..MAXK] of Integer;
  AV, SP, AChord, Max : Integer;
begin
  SP := 0;
  AV := c[1].a;
  AChord := 1;
  Max := 0;
  while True do begin
    {Končí tu nějaký mnohoúhelník?}
    while (SP > 0) and (c[StackC[SP]].b = AV) do begin
      if StackN[SP] > Max then {Je největší?}
        Max := StackN[SP];
      Dec(SP);
      if SP > 0 then
        Inc(StackN[SP]);  {Přidáme vrchol za právě ukončenou tětivu}
    end;
    if AV = n then      {Už jsme prošli celý mnohoúhelník?}
      break;
    {Začíná zde nějaký mnohoúhelník?}
    while (AChord <= k) and (AV = c[AChord].a) do begin
      Inc(SP);
      StackC[SP] := AChord;
      StackN[SP] := 1;
      Inc(AChord);
    end;
    {Začíná dříve nějaká tětiva, než končí jiná?}
    if (AChord <= k) and (c[AChord].a < c[StackC[SP]].b) then begin
      Inc(StackN[SP], c[AChord].a - AV);
      AV := c[AChord].a;
    end
    else begin  {Nějaká tětiva nejdříve končí}
      Inc(StackN[SP], c[StackC[SP]].b - AV);
      AV := c[StackC[SP]].b;
    end;
  end;
  FindMax := Max;
end;

begin
```

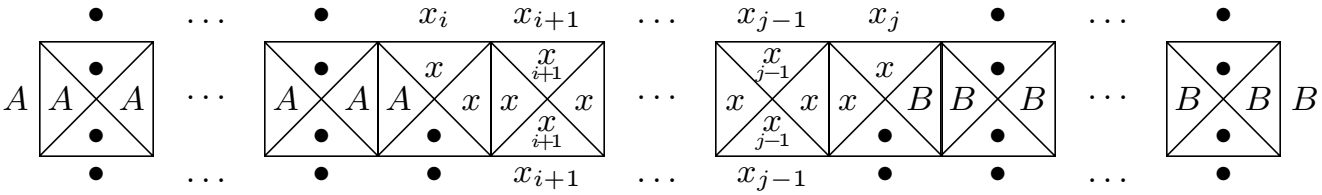
```
  ReadInp;       {Načte vstup}
  SortCut(1, k); {Setřídíme pole se střihy}
  {Nalezne největší část a vypíše ji}
  WriteLn('Nejvetsi cast ma ', FindMax(n, 1, k), ' vrcholu.');
end.
```

P-II-4 *(Dlaždice)*

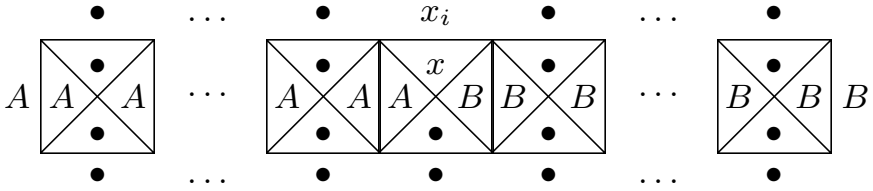
Využijeme jednoduché pozorovanie: Postupnosť $x_1, \ldots, x_n$ je symetrická práve vtedy, keď $x_1 = x_n$ a postupnosť $x_2, \ldots, x_{n-1}$ je symetrická. Jednoprvková i prázdná postupnosť sú vždy symetrické.
Zostrojíme dlaždicový program, ktorý bude pripúšťať len také vydláždenia, v ktorých každý riadok overí rovnosť krajných prvkov postupnosti a odstráni ich (prepíše na farbu •), pričom posledný riadok akceptuje prázdnu alebo jednoprvkovú postupnosť. Takýto program odpovedá *áno* práve na symetrické postupnosti: Ak je postupnosť $x_1, \ldots, x_n$ symetrická, prvý riadok z nej odstráni x_1 a x_n, druhý x_2 a x_{n-1}, atď, až posledný riadok akceptuje jej prostredný prvok (ak mala nepárnu dĺžku) alebo prázdnu postupnosť (ak mala párnu dĺžku). Ak program postupnosť akceptuje, tak podľa prvého riadku je $x_1 = x_n$, podľa druhého $x_2 = x_{n-1}$ atď, teda zadaná postupnosť je symetrická. Preto náš program rieši zadanú úlohu so zložitosťou $O(n)$.
Použijeme nasledovnú sadu dlaždíc:

$$T = \left\{ \begin{array}{ccccccc} \bullet & x & y & x & \bullet & x \\ A\ A & A\ x & x\ x & x\ B & B\ B & A\ B \\ \bullet & \bullet & y & \bullet & \bullet & \bullet \end{array} \; ; \; 0 \le x, y \le 9 \right\},$$

ľavý okraj steny má farbu A, pravý farbu B a dolný farbu •.
Každý korektne vydláždený riadok musí vyzerať buď takto:



(to zodpovedá kontrole a odstráneniu krajných prvkov postupnosti) alebo takto:



(taký riadok akceptuje ľubovoľnú jednoprvkovú postupnosť; prázdna postupnosť – taká, ktorej všetky prvky už boli prepísané na • – je akceptovaná priamo dolným okrajom steny).

Poznámka:

Lepšia časová zložitosť ako lineárna sa nedá dosiahnuť. Myšlienka dôkazu:

Z de nície vydláždenia vieme, že stenu párnej šírky vieme vydláždiť práve vtedy, ak sa dá vydláždiť jej ľavá aj jej pravá polovica tak, aby v každom riadku mali dlaždice, ktorými sa tieto polovice dotýkajú, rovnakú farbu spoločnej hrany (týmto hranám budeme hovoriť *prostredný stĺpec*). Pre každý začiatok postupnosti $x_1, \ldots, x_{n/2}$ však existuje práve jedno doplnenie prvkami $x_{n/2+1}, \ldots, x_n$ také, že $x_1, \ldots, x_n$ je symetrická postupnosť. Keby nejakým dvom rôznym $x_1, \ldots, x_n$ a $y_1, \ldots, y_n$ zodpovedalo rovnaké ofarbenie prostredného stĺpca, potom by sa ale dala vydláždiť aj stena s postupnosťou $x_1, \ldots, x_{n/2}, y_{n/2+1}, \ldots, y_n$ (ľavá časť po prostredný stĺpec rovnako, ako pre $x_1, \ldots, x_n$, pravá časť ako pre $y_1, \ldots, y_n$). To je ale spor, lebo táto postupnosť nie je symetrická. Preto rôznych ofarbení prostredného stĺpca (tých je $\leq f^h$, kde f je počet farieb, vyskytujúcich sa na dlaždiciach, h je maximálna výška steny, teda zložitosť programu) musí byť aspoň toľko, koľko je možných postupností (tých je $10^{n/2}$), a preto musí byť

$$h \geq \log_b 10^{n/2} = n \cdot \frac{\log_b 10}{2}$$

To ale znamená, že zložitosť ľubovoľného dlaždicového programu, riešiaceho našu úlohu, musí byť aspoň lineárna.

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY
50. ROČNÍK MATEMATICKEJ OLYMPIÁDY
Vzorové riešenia 2. kola kategórie P
Vydala IUVENTA – zariadenie pre voľný čas detí, mládeže i dospelých MŠ SR
pre vnútornú potrebu Ministerstva školstva SR
Programom TEX sadzbu pripravil Michal Forišek
Autori príkladov: Jan Kára
Daniel Kráľ
Martin Mareš
© Slovenská komisia matematickej olympiády, 2000