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

51. ročník matematickej olympiády

Riešenia úloh I. kola kategórie P

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

P-I-1

Najskôr prevedieme úlohu do grafovej terminológie. Pojmom *graf* budeme nazývať usporiadanú dvojicu $G = (V, E)$, kde $E \subseteq V \times V$. Prvky množiny V sa nazývajú *vrcholy grafu* a prvky E (čo sú vlastne usporiadané dvojice vrcholov) *hrany grafu*. Vrcholy $u, v \in V$ sú spojené hranou práve vtedy, keď $(u, v) \in E$. *Bipartitný graf* je taký graf, v ktorom môžeme vrcholy rozdeliť na dve disjunktné množiny R a S tak, aby každá hrana spájala vrchol z množiny R s vrcholom z množiny S. Štvorcovú maticu A núl a jednotiek rozmerov $n \times n$ môžeme chápať aj ako reprezentáciu bipartitného grafu G s $2n$ vrcholmi, kde vrcholy $r_1, r_2, \ldots r_n$ zodpovedajú riadkom a vrcholy $s_1, s_2, \ldots, s_n$ zodpovedajú stĺpcom matice. Vrcholy r_i a s_j sú spojené hranou práve vtedy, keď prvok $A[i, j] = 1$. Hranu medzi r_i a s_j budeme značiť (r_i, s_j).

Hovoríme, že bipartitný graf má *úplné párovanie*, ak sa jeho vrcholy dajú usporiadať do dvojíc tak, že v každej dvojici je jeden vrchol z množiny S a jeden vrchol z množiny R a tieto dva vrcholy sú spojené hranou. Navyše, každý vrchol sa musí nachádzať práve v jednej takejto dvojici.

Ukážeme, že ak bipartitný graf G má úplné párovanie, tak v našej matici A možno preusporiadať riadky a stĺpce takým spôsobom, aby na diagonále boli samé jednotky. Ak párovanie obsahuje dvojice $(r_{i_1}, s_{j_1}), (r_{i_2}, s_{j_2}), \ldots, (r_{i_n}, s_{j_n})$, potom stačí riadky usporiadať v poradí $i_1, i_2, \ldots, i_n$ a stĺpce v poradí $j_1, j_2, \ldots, j_n$. Označme takto preusporiadanú maticu A'. Platí $A'[k, k] = A[i_k, j_k]$. Keďže medzi vrcholmi r_{i_k} a s_{j_k} v grafe G vedie hrana, platí, že $A[i_k, j_k] = 1$, a preto matica A' má na diagonále samé jednotky.

Naopak, ak sa dá matica A preusporiadať tak, aby mala na diagonále samé jednotky, tak v grafe G existuje úplné párovanie. Nech v preusporiadanej matici A' sú riadky v poradí $i_1, i_2, \ldots, i_n$ a stĺpce v poradí $j_1, j_2, \ldots, j_n$. Vieme, že pre všetky k platí $A'[k, k] = 1$, a preto aj $A[i_k, j_k] = 1$. Preto v grafe G dvojice $(r_{i_1}, s_{j_1}), (r_{i_2}, s_{j_2}), \ldots, (r_{i_n}, s_{j_n})$ tvoria úplné párovanie. Dokázali sme teda nasledujúce tvrdenie.

Tvrdenie. Matica A sa dá transformovať na tvar s jednotkami na diagonále práve vtedy, keď existuje úplné párovanie v grafe G.

Ak teda chceme zistiť, ako maticu A pretransformovať na tvar s jednotkami na diagonále, vypočítame najprv úplné párovanie v grafe G. Ak také párovanie existuje, preusporiadame riadky a stĺpce podľa postupu uvedeného vyššie. Ak také párovanie neexistuje, maticu nemožno transformovať. Jediným problémom zostáva ukázať, ako také úplné párovanie nájsť.

Pripomeňme, že v úplnom párovaní sú v každej dvojici vrcholy spojené hranou. Preto môžeme úplné párovanie chápať aj ako množinu hrán M takú, že každý vrchol grafu je koncovým vrcholom práve jednej hrany z M. Podobne môžeme zadefinovať *párovanie* ako množinu hrán M takú, že každý vrchol grafu je koncovým vrcholom najviac jednej hrany z M. To znamená, že v párovaní, ktoré nie je úplné, nie všetky vrcholy musia byť zaradené v niektorej dvojici. Ukážeme teraz algoritmus, ako nájsť v bipartitnom grafe maximálne párovanie, t.j. párovanie s najväčším počtom hrán.

Maximálne párovanie budeme hľadať postupne. Začneme s prázdnym párovaním $M = \emptyset$ a v každom kroku zvýšime počet párovacích hrán o jedna. Ak sa nám v niektorom kroku nepodarí zvýšiť počet hrán v párovaní, vyhlásime aktuálne párovanie za maximálne a skončíme.

Zvyšovať počet hrán v párovaní budeme pomocou takzvaných zlepšujúcich ciest. Uvažujme ľubovoľné pevne zvolené párovanie M. *Alternujúca cesta* pre párovanie M je postupnosť vrcholov $r_{i_1}, s_{j_1}, r_{i_2}, s_{j_2}, \ldots, r_{i_k}, s_{i_k}$, ktorá začína nespárovaným riadkovým vrcholom, končí stĺpcovým vrcholom, každá dvojica po sebe nasledujúcich vrcholov je spojená hranou a striedajú sa párovacie a nepárovacie hrany (prvá hrana (r_{i_1}, s_{j_1}) je nepárovacia). *Zlepšujúca cesta* je alternujúca cesta, ktorá končí nespárovaným vrcholom.

Všimnite si, že zlepšujúca cesta $P = r_{i_1}, s_{j_1}, r_{i_2}, s_{j_2}, \ldots, r_{i_k}, s_{i_k}$ pozostáva z $k-1$ párovacích a k nepárovacích hrán. Majme teda párovanie M a zlepšujúcu cestu P. Ak všetky párovacie hrany v P zmeníme na nepárovacie a naopak, dostaneme nové párovanie M', ktoré má o jednu hranu viac (uvedomte si, že M' naozaj spĺňa podmienky párovania).

(plné čiary sú párovacie, čiarkované čiary sú nepárovacie)

Dokážme teraz, že našim postupom vždy dostaneme maximálne párovanie, t.j. že sa nám nestane, že neexistuje zlepšujúca cesta pre párovanie, ktoré nie je maximálne.

Tvrdenie. Ak párovanie M nie je maximálne, existuje preňho zlepšujúca cesta v G.

Dôkaz tvrdenia. Nech M je párovanie, ktoré nie je maximálne. Keďže M nie je maximálne, musí existovať párovanie M', ktoré obsahuje viac hrán ako M. Hrany patriace do M, avšak nie do M' nazvime *modré* a hrany patriace do M' avšak nie do M nazvime *červené*. Keďže $|M'| > |M|$, červených hrán musí byť viac ako modrých.

Uvažujme graf G' tvorený všetkými modrými a červenými hranami. Každý vrchol v G' susedí s najviac jednou modrou a jednou červenou hranou. Teda každý súvislý komponent grafu G' musí byť buď kružnica alebo cesta, pričom na každej kružnici (ceste) sa striedajú červené a modré hrany. To znamená, že každá kružnica obsahuje rovnako veľa modrých a červených hrán; počty červených a modrých hrán na každej ceste sa líšia najviac o 1. Keďže červených hrán je viacej ako modrých, musí existovať komponent P, ktorý obsahuje viac červených hrán ako modrých. Takýto komponent musí byť cesta, ktorá začína aj končí červenou hranou. Teda cesta P obsahuje nepárny počet hrán. Z toho vyplýva, že jeden z jej koncov musí byť riadkový vrchol (nazvime ho r_i) a druhý stĺpcový vrchol (nazvime ho s_j). Je ľahké presvedčiť sa, že keďže r_i resp. s_j je nespárovaný riadkový resp. stĺpcový vrchol a každá druhá hrana patrí do párovania M, P je zlepšujúca cesta pre M.

Implementácia. Graf G reprezentujeme samotnou maticou, t.j. dvojrozmerným poľom A. Pre každý vrchol r_i (s_j) si v poli `par_r` (`par_s`) pamätáme, či je spárovaný s nejakým vrcholom a ak áno, číslo tohoto vrchola. Začneme so všetkými vrcholmi nespárovanými. Funkcia `Zlepsi`

hľadá zlepšujúcu cestu a ak ju nájde, použije ju na zlepšenie aktuálneho párovania. Hľadanie je implementované prehľadávaním do šírky zo všetkých vrcholov r_i, ktoré ešte nie sú spárované. Ak nájde alternujúcu cestu do nespárovaného vrchola s_j, prehľadávanie skončí. V tomto okamihu je potrebné prejsť po nájdenej ceste z s_j späť do r_i, zmeniť párovacie hrany na nepárovacie a naopak, čo znamená aktualizovanie záznamov v `par_r` (`par_s`) pre všetky vrcholy na nájdenej ceste. Funkcia `Zlepsi` je volaná v cykle, kým je možné zvyšovať počet hrán v párovaní. Ak vylepšenie párovania úspešne zbehne n krát, máme úplné párovanie, ktoré použijeme na preusporiadanie matice do požadovaného tvaru. V opačnom prípade podáme správu o tom, že matica sa usporiadať nedá. Časová zložitosť: každé volanie `Zlepsi` zbehne v čase $O(n^2)$, spolu potrebujeme nanajvýš n volaní, čo dáva celkový čas $O(n^3)$. Na uloženie matice A potrebujeme pamäť $O(n^2)$.

```
program P_I_1;

const MAXN = 100;
var f, g : text;
    A : array[1..MAXN, 1..MAXN] of integer;
    par_r, par_s : array[1..MAXN] of integer;
    N : integer;

function Zlepsi : boolean;
{ak sa dá, zvýši počet párov o 1}
var buf, back : array[0..MAXN] of integer;
    zbuf, kbuf, vi, vj, i, j, tmp : integer;
    nasiel : boolean;
Begin
  zbuf := 0; kbuf := 0;
  for i := 1 to N do
    if par_r[i] <= 0 then begin
      buf[kbuf] := i; Inc(kbuf);
    end;
  for j := 1 to N do back[j] := 0;

  nasiel := false;
  while (zbuf < kbuf) and not nasiel do begin
    vi := buf[zbuf]; Inc(zbuf);
    for j := 1 to N do begin
      if nasiel then break;
      if (A[vi, j] = 0) or (j = par_r[vi]) or (back[j] > 0) then continue;
      if par_s[j] <= 0 then begin
      {našli sme zlepšujúcu cestu}
        nasiel := true;
        vj := j;
        while true do begin
          par_s[vj] := vi;
          tmp := par_r[vi];
          par_r[vi] := vj;
          vj := tmp;
          if vj <= 0 then break;
```

```
          vi := back[vj];
        end;
      end else begin
      {pokračujeme v hľadaní}
        back[j] := vi;
        buf[kbuf] := par_s[j];
        Inc(kbuf);
      end;
    end;
  end;
  Zlepsi := nasiel;
End;

procedure Uprac_stlpce;
var i, j, tmp, ii : integer;
Begin
  for i := 1 to n do
    while par_r[i] <> i do begin
      ii := par_r[i];
      for j := 1 to n do begin
        tmp := A[i, j]; A[i, j] := A[ii, j]; A[ii, j] := tmp;
      end;
      par_r[i] := par_r[ii];
      par_r[ii] := ii;
    end;
End;

var i, j : integer;
    ok : boolean;
Begin
  Assign(f, 'matica.in'); Reset(f);
  Assign(g, 'matica.out'); ReWrite(g);
  ReadLn(f, N);
  for i := 1 to N do for j := 1 to N do
    Read(f, A[i, j]);

  {párovanie}
  for i := 1 to N do begin
    ok := Zlepsi;
    if not ok then break;
  end;

  if not ok then WriteLn(g, 'Maticu nemozno transformovat.')
  else begin
    Uprac_stlpce;
    for i := 1 to n do
      for j := 1 to n do begin
        Write(g, A[i, j]);
        if j < n then Write(g, ' ') else WriteLn(g);
```

```
    end;
  end;

  Close(g); Close(f);
End.
```

P-I-2

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. Ak navyše $v_0 = v_k$, takýto sled nazveme *uzavretý*. V reči grafov, úlohou bolo v danom súvislom grafe G nájsť uzavretý sled, ktorý prejde každú hranu grafu práve dvakrát.

Ukážeme algoritmus, ktorý pre ľubovoľný súvislý graf nájde požadovaný sled (tento algoritmus je teda vlastne dôkazom, že trasa pre robota vždy existuje). Naše riešenie je založené na prehľadávaní do hĺbky. O každom vrchole si budeme pamätať, či sme ho už v priebehu algoritmu niekedy navštívili. Pre každý navštívený vrchol v, nech `rodic`$[v]$ označuje vrchol, z ktorého robot do v prvýkrát prišiel. Vrchol `rodic`$[v]$ je *rodičom* vrchola v; podobne v je *potomkom* vrchola `rodic`$[v]$. Pre vrchol 1 nie je `rodic`[1] definovaný.

Prehľadávanie do hĺbky začína z vrchola 1 a funguje takto:

1. Označ aktuálny vrchol v ako navštívený.
2. Postupne kontroluj všetky hrany idúce z v a vždy keď nájdeš takú hranu, ktorá vedie do ešte nenavštíveného vrchola u, prejdi robotom do vrchola u a pokračuj rekurzívne v prehľadávaní z u (v sa stane rodičom u).
3. Keď sú všetky hrany z v skontrolované a $v \neq 1$, vráť sa do vrchola `rodic`$[v]$ (a pokračuj v kontrolovaní jeho hrán). Ak $v = 1$, skonči.

Uvažujme množinu hrán, po ktorých robot prvýkrát prišiel do nejakého vrchola; sú to práve hrany (`rodic`$[v], v$), pre tie v, pre ktoré je `rodic`$[v]$ definovaný. Nazvime tieto hrany *stromové* (nazývajú sa tak, lebo tieto hrany tvoria súvislý graf neobsahujúci kružnicu, nazývaný tiež strom). Každú stromovú hranu (u, v), kde u je rodičom v, prejde robot pri prehľadávaní práve dvakrát – najprv z vrchola u prvýkrát navštívi vrchol v a druhýkrát pri návrate z vrchola v do vrchola u. Pri prehľadávaní robot nikdy neprejde po nestromovej hrane (keď prechádza po hrane (u, v), buď navštevuje vrchol v prvýkrát, čím sa hrana (u, v) stane stromovou, alebo sa vracia z vrchola u do vrchola v a to vždy po stromovej hrane). Keďže náš graf G je súvislý, robot pri prehľadávaní navštívi všetky vrcholy. Keby totiž existoval nenavštívený vrchol, musel by existovať taký navštívený vrchol u a nenavštívený vrchol v, že u a v sú spojené hranou. To však nie je možné, lebo algoritmus prehľadávania sa z vrchola u do `rodic`$[u]$ nevráti, kým nie sú všetci susedia u navštívení.

Prehľadávanie do hĺbky navštívi každý vrchol a prejde každú stromovú hranu práve dvakrát; ostáva nám rozhodnúť kedy a ako bude robot prechádzať nestromové hrany. Počas prehľadávania je jednoduché detekovať nestromové hrany. Ak sa robot práve nachádza vo vrchole u a kontroluje hranu (u, v), táto hrana je nestromová, ak vrchol v už bol navštívený a u nie je rodičom v ani v nie je rodičom u. Ak robot nachádzajúci sa vo vrchole u narazí na nestromovú hranu (u, v), môže ju vyčistiť prejdením do v a späť za predpokladu, že ňou už predtým neprešiel. Preto si pre každú hranu budeme pamätať, či už bola vyčistená alebo nie.

Implementácia. Prehľadávanie do hĺbky je implementované rekurzívnou procedúrou `Prehladaj`. Keďže rekurzia si pre aktuálny vrchol automaticky pamätá jeho rodiča, nie je nutné

explicitne udržiavať záznamy `rodic`$[v]$. Vo vzorovom programe graf reprezentujeme maticou $n \times n$, z čoho vyplýva časová aj pamäťová zložitosť programu $O(n^2)$. Efektívnejšou reprezentáciou hrán pomocou spájaného zoznamu je možné dosiahnuť časovú aj pamäťovú zložitosť $O(m+n)$, kde m je počet hrán grafu, avšak za cenu mierne zložitejšieho programu.

Poznámka. Úlohu bolo možné riešiť aj pomocou algoritmov na hľadanie eulerovského ťahu (t.j. sledu, ktorý obsahuje každú hranu práve raz). Stačí náš graf pozmeniť tak, že každú hranu skopírujeme dvakrát a v tomto grafe nájsť eulerovský ťah (ten vždy existuje, lebo každý vrchol v novom grafe bude mať párny stupeň). Tento typ riešenia nebudeme podrobnejšie popisovať.

```
program P_I_2;

const MAXN = 100;            {max. počet vrcholov      }
      HR_HRANA = 1;          {typy hrán - neprejdená,  }
      HR_STROMOVA = 2;       {     stromová a          }
      HR_PREJDENA = 3;       {     nestromová prejdená }

var f, g : text;
    n, m : integer;
    hr : array[1..MAXN, 1..MAXN] of integer;
    bol : array[1..MAXN] of boolean;

procedure Prehladaj(v : integer);
var i : integer;
Begin
  WriteLn(g, v);
  bol[v] := true;
  for i := 1 to n do begin
    if hr[v][i] <> 1 then continue;
    if bol[i] then begin     {nestromová neprejdená hrana}
      WriteLn(g, i);
      hr[v, i] := HR_PREJDENA;
      hr[i, v] := HR_PREJDENA;
    end else begin           {stromová neprejdená hrana}
      hr[v, i] := HR_STROMOVA;
      hr[i, v] := HR_STROMOVA;
      Prehladaj(i);
    end;
    WriteLn(g, v);
  end;
End;                         {Prehľadaj}

var i, j, aa, bb : integer;
Begin
  Assign(f, 'rury.in'); Reset(f);
  Assign(g, 'rury.out'); ReWrite(g);
  ReadLn(f, n, m);
  for i := 1 to n do for j := 1 to n do hr[i, j] := 0;
  for i := 1 to m do begin
    ReadLn(f, aa, bb);
```

```
    hr[aa, bb] := HR_HRANA; hr[bb, aa] := HR_HRANA;
  end;
  for i := 1 to n do bol[i] := false;

  Prehladaj(1);

  Close(g); Close(f);
End.
```

P-I-3

Dokážme najprv, že spravodlivá priamka vždy existuje. Vezmime ľubovoľnú orientovanú priamku idúcu cez bod $[0, 0]$ a nech x je rozdiel počtu bielych bodov naľavo a čiernych bodov napravo. Ak je x nula, priamka je spravodlivá. Ak nie, budeme priamku otáčať okolo bodu $[0, 0]$. Vždy keď priamka prejde cez biely alebo čierny bod, hodnota x sa zníži alebo zvýši o jedna. Keď priamku otočíme presne o 180°, naša sledovaná hodnota prešla z počiatočného x na $-x$ (lebo všetky body, ktoré boli pôvodne naľavo, sú teraz napravo a naopak). To znamená, že hodnota musela byť aspoň pri jednej polohe priamky nulová. V tejto polohe je priamka spravodlivá.

Budeme vravieť, že bod A je naľavo od bodu B, ak je naľavo od orientovanej priamky idúcej z $[0, 0]$ do A. Základom algoritmu je nasledujúce pozorovanie. Vezmime spravodlivú priamku p. Nech A je prvý bod, na ktorý by sme narazili, ak by sme priamku p otáčali v smere hodinových ručičiek. Bod A a všetky body, ktoré sú od neho napravo, sú v jednej polrovine určenej priamkou p. Body, ktoré sú od A naľavo, sú v druhej polrovine. Vidíme teda, že nezáleží na konkrétnej polohe priamky p, rozdelenie bodov je určené polohou "hraničného" bodu A.

Stačí teda zobrať každý bod ako kandidáta na bod A, zrátať počet bielych a čiernych bodov naľavo a napravo a skontrolovať, či počty spĺňajú podmienky spravodlivej priamky. Ak nájdeme úspešného kandidáta, spravodlivú priamku zostrojíme tak, že priamku idúcu cez bod otočíme kúsok proti smeru hodinových ručičiek tak, aby sme neprešli cez žiaden iný bod. V praxi sa to dá spraviť tak, že nájdeme prvý bod X, cez ktorý by sme pri takom otáčaní prešli a priamku vedieme stredom medzi kandidátom A a bodom X. Takto dostaneme algoritmus s časovou zložitosťou $O(n^2)$ (pre každého kandidáta zisťujeme polohu každého bodu vzhľadom na tohto kandidáta).

Algoritmus však možno zefektívniť tým, že si body najprv utriedime v smere hodinových ručičiek okolo bodu $[0, 0]$. Kandidátov na hraničný bod A budeme skúšať v tomto utriedenom poradí. Keď sa presunieme z kandidáta $i - 1$ na kandidáta i, nemusíme už pre každý bod zisťovať, či je od bodu i naľavo alebo napravo, lebo veľa bodov zostane v tej istej polrovine. Budeme si teda v každom kroku udržovať počet bielych a čiernych bodov v pravej polrovine a index j ukazujúci na posledný bod v pravej polrovine v smere hodinových ručičiek. Keď se presunieme na kandidáta i, stačí bod $i - 1$ presunúť do ľavej polroviny (t.j. odpočítať z počtu bodov príslušnej farby, ktoré sú v pravej polrovine) a potom posúvať index j v smere hodinových ručičiek, kým nenájdeme prvý bod, ktorý je naľavo od bodu i. Všetky body, cez ktoré sme s indexom j prešli, sa presunú z ľavej polroviny do pravej a treba ich teda pripočítať k počtu bodov príslušnej farby. Keď pri posúvaní bodu j prídeme na koniec poľa, pokračujeme zase cyklicky od začiatku. Netreba zabudnúť ošetriť okrajové prípady, keď napríklad všetky body sú od bodu j naľavo alebo napravo.

Utriediť body v smere hodinových ručičiek môžeme v čase $O(n \log n)$ niektorým so štandard-

ných triediacich algoritmov. Iba namiesto bežného porovnania si potrebujeme napísať funkciu, ktorá pre dva body určí, ktorý z nich má väčší uhol. V priloženom programe sme pre jednoduchosť použili triedenie pracujúce v čase $O(n^2)$, to však možno ľahko nahradiť iným.

Pre prvý bod musíme prejsť všetky body a zistiť, ktoré sú vľavo (v čase $O(n)$). Pre každý ďalší bod už len budeme posúvať indexy i a j a sledovať iba body, cez ktoré prechádzame. Cez každý bod prejdeme každým indexom najviac raz a teda celkový čas posúvania pre všetky kandidátske body spolu bude $O(n)$. Celková časová zložitosť algoritmu je teda $O(n \log n)$ a pamäťová zložitosť je $O(n)$.

```
program p_I_3;
const napriamke = 0; napravo = 1; nalavo = 2; {smery}
      biela = 0; cierna = 1;                  {farby}
type bod = record x, y : longint; farba : integer; end;
var n : integer;          {počet bielych/čiernych bodov}
    m : integer;                {celkový počet bodov}
    a : array[1..1000] of bod;            {pole bodov}
     {pocet bielych a čiernych bodov napravo do priamky}
    pocet : array[biela..cierna] of integer;
    start : bod;            {štartovací bod triedenia}

function poloha(a, b : bod) : integer;
{Zisti, kde je bod b vzhľadom na priamku cez body (0,0),a}
var vektsucin : longint;
begin
   vektsucin := a.x * b.y - b.x * a.y;
   if vektsucin > 0 then poloha := nalavo
   else if vektsucin < 0 then poloha := napravo
   else poloha := napriamke
end; {poloha}

function mensi(var a, b : bod) : boolean;
{Vráti, či bod a je menší ako bod b v našom usporiadaní}
var pol1, pol2, pol3 : integer;
begin
   pol1 := poloha(start, a);
   pol2 := poloha(start, b);
   pol3 := poloha(a, b);
   mensi := (pol1 < pol2) or ((pol1 = pol2) and (pol3 = napravo));
end; {mensi}

procedure tried;
{Utriedi body pomocou porovnania „mensi“}
var i, j, min : integer;
   pom : bod;
begin
   for i := 1 to m - 1 do begin
      min := i;                {nájdi minimum v a[i..m]}
      for j := i + 1 to m do begin
         if mensi(a[j], a[min]) then min := j;
```

```
        end;
        {ulož minimum do a[i]}
        pom := a[i]; a[i] := a[min]; a[min] := pom;
    end;
end; {tried}

procedure nacitaj;
{Načíta dáta zo súboru}
var t : text;
    i : integer;
begin
    assign(t, 'priamka.in');   {otvor súbor}
    reset(t);
    read(t, n);          {počet bielych/čiernych bodov}
    m := 2 * n;          {celkový počet bodov}
    for i := 1 to m do begin {čítaj body}
        read(t, a[i].x, a[i].y);
        if i <= n then a[i].farba := biela
        else a[i].farba := cierna;
    end;
    close(t);          {zavri súbor}
end; {nacitaj}

function prva_priamka : integer;
{Nájdi priamku idúcu tesne pred bod a[1],
 vráť posledný bod napravo, inicializuj pole pocet}
var i : integer;
begin
    pocet[biela] := 0;   {vynuluj počty}
    pocet[cierna] := 0;
    i := 1;              {kým sme napravo, zvyšuj pole pocet}
    while (i <= m) and (poloha(a[1], a[i]) in [napravo, napriamke]) do begin
        pocet[a[i].farba] := pocet[a[i].farba] + 1;
        i := i + 1;
    end;
    prva_priamka := i - 1;
end; {prva_priamka}

function dalsia_priamka(i, j : integer) : integer;
{Posunie priamku spred bodu a[i-1] pred bod a[i]}
var koniec : boolean;
    stare_j : integer;
begin
    {a[i-1] sa posunie doľava - odčítaj ho}
    pocet[a[i - 1].farba] := pocet[a[i - 1].farba] - 1;
    koniec := false;
    repeat     {pričítaj body, ktoré sa posunú doprava}
        stare_j := j;
        j := j + 1;
```

```
      if j > m then j := 1;
      if (poloha(a[i], a[j]) in [napravo, napriamke]) then begin
         pocet[a[j].farba] := pocet[a[j].farba] + 1;
      end
      else koniec := true;
   until koniec or (j = i);
   dalsia_priamka := stare_j;
end; {dalsia_priamka}

procedure vypis(i, j : integer);
{Vypíše výsledok do súboru}
var t : text;
    b1, b2 : bod;
begin
   {b1 je a[i] premietnuté podľa (0,0)}
   b1.x := -a[j].x; b1.y := -a[j].y;
   {b2 je posledný bod pred a[i]}
   if i > 1 then b2 := a[i - 1]
   else b2 := a[m];
   {nájdi ten z b1, b2, ktorý je viac vpravo}
   if (poloha(a[i], b2) = nalavo) and (poloha(b1, b2) = napravo) then
      b1 := b2;

   assign(t, 'priamka.out');    {otvor súbor}
   rewrite(t);
   {výsledný bod je stred úsečky medzi a[i] a b1}
   writeln(t, (a[i].x + b1.x)/2.0 : 0 : 1, ' ', (a[i].y + b1.y)/2.0 : 0 : 1);
   close(t);            {zatvor súbor}
end; {vypis}

var i, j : integer;
begin {hlavný program}
   nacitaj;       {načítaj vstup}
   start := a[1];  {od tohoto bodu začneme triediť}
   tried;         {utrieď proti smeru hodinových ručičiek}
   i := 1; j := prva_priamka;  {nájdi prvú priamku}
   {kým nenájdeš čo potrebuješ, skúšaj ďalšiu priamku}
   while pocet[biela] <> n - pocet[cierna] do begin
      i := i + 1;
      j := dalsia_priamka(i, j);
   end;
   vypis(i, j);   {vypíš do výstupného súboru}
end.
```

P-I-4

Časť a. Najskôr popíšme formát vstupného súboru, aký sme si zvolili (váš program samozrejme môže používať iný formát). Vstupný súbor obsahuje na prvom riadku počet vodičov a počet komparátorov, na druhom riadku vstupy siete od vrchného vodiča po spodný a zvyšné riadky

obsahujú popis siete. Každý komparátor je daný dvoma číslami zapísanými na jednom riadku. Tieto čísla určujú vrchný a spodný vodič spojený komparátorom. Vodiče sú číslované zhora nadol začínajúc od jedna. Komparátory sú zadané v poradí zľava doprava. Jednotlivé vrstvy siete sú oddelené riadkom obsahujúcim dve nuly.

Program uvedený v tomto vzorovom riešení pre jednoduchosť zobrazuje výpočet siete semigraficky. Najprv načíta vstup, pričom všetky komparátory a oddeľovače vrstiev uloží do jedného pola. Potom vykresľuje samotnú sieť zľava doprava. Zakaždým, keď vykreslí komparátor, odsimuluje jeho činnosť na aktuálnom stave vodičov. To znamená, že porovná príslušné dve hodnoty v poli, a ak sú v opačnom poradí, tak ich vymení. Po skončení každej vrstvy (t.j. vždy, keď narazí na oddeľovač v zozname komparátorov) program vypíše aktuálny stav hodnôt na vodičoch.

Jediný problém, ktorý zostáva vyriešiť, je ako umiestniť jednotlivé komparátory v jednej vrstve do stĺpcov, tak aby sa žiadne dva komparátory v jednom stĺpci neprekrývali. Náš program si preto pamätá, ktoré z vodičov sú v aktuálnom stĺpci už obsadené komparátorom. Ak sa ďalší komparátor prekrýva s niektorým komparátorom v aktuálnom stĺpci, začneme nový stĺpec a komparátor umiestnime do nového stĺpca.

Celková časová zložitosť nášho programu je úmerná veľkosti nakresleného obrázku, t.j. $O(nk)$, kde n je počet vodičov a k je počet stĺpcov, v ktorých sú komparátory vykreslené. Na záver uveďme príklad vstupu a výstupu z nášho programu (ide o sieť uvedenú v študijnom texte).

Vstup:

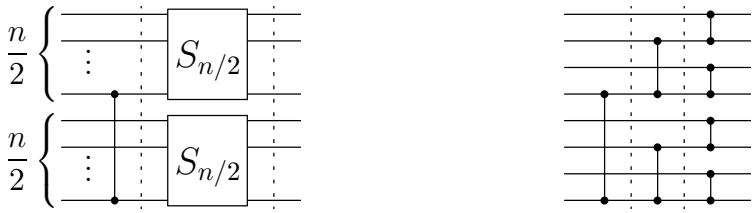
1. časť:
```
4 5
4 1 2 3
1 4
2 3
0 0
```

2. časť:
```
1 2
3 4
0 0
2 3
0 0
```

Výstup:
```
4--*-----3--*--1-----1
   |        |
1--|--*--1--*--3--*--2
   |  |           |
2--|--*--2--*--2--*--3
   |        |
3--*-----4--*--4-----4
```

Časť b. Sieť zostrojíme rekurzívne. Označme S_n sieť, ktorá úlohu rieši pre n vstupov. Sieť S_1 neobsahuje žiaden komparátor, lebo máme iba jeden vstup a ten je utriedený. Predpokladajme, že $n > 1$. Prvá vrstva siete obsahuje iba jeden komparátor – medzi vodičmi n a $n/2$. Potom rozdelíme n vodičov na dve polovice – hornú a dolnú a na každú polovicu použijeme sieť $S_{n/2}$. Tieto dve podsiete budú pracovať paralelne.

Konštrukcia siete S_n je zobrazená na nasledujúcom obrázku vľavo, vpravo je príklad výslednej siete pre $n = 8$.



Uvedomme si najprv, prečo takto zostavená sieť rieši zadanú úlohu. Označme vstupné hodnoty siete $x_1, x_2, \ldots, x_n$. Môžu nastať dva prípady podľa toho, či je x_n menšie alebo väčšie ako $x_{n/2}$. Predpokladajme najprv, že $x_n \geq x_{n/2}$. Vtedy komparátor v prvej vrstve nevymení vstupné hodnoty. Nakoľko $x_n \geq x_{n/2}$, prvok x_n patrí v usporiadanom poradí na niektorý zo spodnej polovice vodičov.

Po prvom kroku výpočtu teda platí, že horná polovica je celá utriedená a obsahuje $n/2$ najmenších prvkov zo vstupu. Dolná polovica je utriedená s prípadnou výnimkou posledného

prvku a obsahuje $n/2$ najväčších prvkov. Preto vstupy oboch podsietí $S_{n/2}$ spĺňajú podmienku zo zadania, že všetky hodnoty okrem poslednej majú byť na vstupe utriedené (to nevylučuje prípad, že aj posledná hodnota bude utriedená). Teda na výstupe budú obe polovice v utriedenom poradí. Keďže už po prvom kroku každá polovica obsahovala tie správne prvky, aj celá postupnosť je v utriedenom poradí.

Druhý prípad nastane, ak $x_n < x_{n/2}$. Vtedy komparátor vymení hodnoty. Vieme, že x_n patrí niekde do hornej polovice prvkov a dostane sa na najspodnejší vodič hornej polovice. Naopak, posledný prvok z hornej polovice, t.j. $x_{n/2}$, patrí v skutočnosti do dolnej polovice (pri triedení je "vytlačený" prvkom x_n). Prvok $x_{n/2}$ sa ale prvým komparátorom dostane na najspodnejší vodič, t.j. do dolnej polovice. Podobne ako predtým, aj teraz máme po prvom kroku v každej polovici tie prvky, ktoré tam v utriedenom poradí patria. Taktiež obe polovice sú utriedené s prípadnou výnimkou najspodnejšieho prvku. Teda po aplikovaní podsietí $S_{n/2}$ dostaneme dve utriedené polovice, ktoré spolu tvoria utriedenú postupnosť.

Hĺbka rekurzie je v našej konštrukcii $\log_2 n$, lebo každá úroveň rekurzie zmenší počet vodičov na polovicu. V každej úrovni rekurzie pridáme jednu vrstvu, celkový počet vrstiev je teda tiež $\log_2 n$.

Prvá vrstva obsahuje 1 komparátor, v každej ďalšej vrstve sa počet komparátorov zdvojnásobí. Nech počet vstupov je $n = 2^k$. Potom počet použitých komparátorov je $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1 = n - 1$ (súčet geometrickej postupnosti). Naša sieť teda má $O(\log n)$ vrstiev a používa $O(n)$ komparátorov.

```
program P_I_4a;
{program semigraficky zobrazí priebeh výpočtu komparátorovej siete}
uses crt;
const MAXM = 100; {maximálny počet komparátorov}
      MAXN = 100; {maximálny počet vodičov}
var n : integer;   {počet vodičov}
    poloziek : integer;   {počet položiek v poli komparatory}
    komparatory : array[1..2 * MAXM, 1..2] of integer;
                          {zoznam komparátorov a oddeľovačov úrovní}
    stav : array[1..MAXN] of integer;
                          {aktuálny stav vodičov po niekoľkých krokoch}
    plne : array[1..MAXN] of boolean;
                          {či ide cez vodič čiara v aktuálnom stĺpci}

procedure nacitaj;
var inp : text;       {vstupný súbor}
    i, komparatorov, bolo_komparatorov : integer;
begin
   assign(inp, 'siete.in');    {otvor vstupný súbor}
   reset(inp);
   read(inp, n, komparatorov); {čítaj počet vodičov a komparátorov}
   for i := 1 to n do     {načítaj vstupy siete - (stav vodičov)}
      read(inp, stav[i]);

   {načítavaj komparátory a oddeľovače,
    až kým nedosiahneš daný počet komparátorov}
   bolo_komparatorov := 1; poloziek := 0;
   while bolo_komparatorov <= komparatorov do begin
```

```
        poloziek := poloziek + 1;
        read(inp, komparatory[poloziek, 1], komparatory[poloziek, 2]);
        {ak položka bola komparátor, zvýš počet komparátorov}
        if komparatory[poloziek, 1] <> 0 then
            bolo_komparatorov := bolo_komparatorov + 1;
    end;
    {za posledným načítaným komparátorom pridaj oddeľovač}
    poloziek := poloziek + 1;
    komparatory[poloziek, 1] := 0;
    komparatory[poloziek, 2] := 0;
    close(inp);                    {zatvor vstupný súbor}
end; { nacitaj }

procedure prazdne_stlpce(stlpec, kolko : integer);
var i, j : integer;
begin {vypis <kolko> prázdnych stľpcov}
    for i := 1 to n do begin
        gotoxy(stlpec, i * 2 - 1);
        for j := 1 to kolko do write('-');
    end;
end; { prazdne_stlpce }

procedure vypis_stav(stlpec, sirka : integer);
var i : integer;
begin
    for i := 1 to n do begin {vypíš aktuálny stav vodičov}
        gotoxy(stlpec, i * 2 - 1);
        write(stav[i] : sirka); {každé číslo má šírku <sirka>}
    end;
end; { vypis_stav }

function najdi_sirku : integer;
var i, max, sirka, cislo : integer;
begin {najde šírku (počet miest) najširšieho čísla na vstupe}
    max := 1;
    for i := 1 to n do begin
        sirka := 1;
        cislo := stav[i];          {nájdi šírku čísla stav[i]}
        while cislo >= 10 do begin
            sirka := sirka + 1;    {del 10, kým neklesneš pod 10}
            cislo := cislo div 10;
        end;
        if sirka > max then max := sirka; {porovnaj s maximom}
    end;
    najdi_sirku := max;
end; { najdi_sirku }

function je_volno(odkial, pokial : integer) : boolean;
var i : integer;
```

```
    ok : boolean;
begin {zisti, či môžeme zakresliť komparátor do stĺpca}
   ok := true;
   for i := odkial to pokial do begin
      if plne[i] then ok := false;  {ak sa kríži s iným, nemôžeme}
   end;
   je_volno := ok;
end; { je_volno }

procedure spracuj_komparator(odkial, pokial, stlpec : integer);
var i, pom : integer;
begin {vykresli jeden komparátor, zapíš do pola plne, meň stav}
   gotoxy(stlpec, 2 * odkial - 1);    {vykresli horný koniec}
   write('*');
   gotoxy(stlpec, 2 * pokial - 1);    {vykresli dolný koniec}
   write('*');
   for i := 2 * odkial to 2 * pokial - 2 do begin  {spojnica medzi}
      gotoxy(stlpec, i);
      write('|');
   end;
   for i := odkial to pokial do   {zapíš komparátor do pola plne}
      plne[i] := true;
   {vykonaj porovnanie a výmenu danú komparátorom}
   if stav[odkial] > stav[pokial] then begin
      pom := stav[odkial];
      stav[odkial] := stav[pokial];
      stav[pokial] := pom;
   end;
end; { vykresli_komparator }

procedure vykresli;
var i, j, stlpec, sirka : integer;
    novy_stlpec : boolean;
begin {vykresli celý priebeh výpočtu}
   clrscr;              {zmaž obrazovku}
   sirka := najdi_sirku; {nájdi šírku čísla na vodiči}
   vypis_stav(1, sirka); {vypíš vstupy do stĺpca 1}
   stlpec := sirka;      {posledný použitý stĺpec je <sirka>}
   novy_stlpec := true;  {v tomto stĺpci nebol žiaden komparátor}

   for i := 1 to poloziek do begin
      if komparatory[i][1] = 0 then begin   {oddeľovač úrovní}
         prazdne_stlpce(stlpec + 1, 2);      {vykresli vodiče}
         vypis_stav(stlpec + 3, sirka);      {vypíš aktuálny stav}
         stlpec := stlpec + 2 + sirka;
         novy_stlpec := true;      {začneme nový stĺpec komparátorov}
      end
      else begin   {komparátor}
         if novy_stlpec
```

```
            or not je_volno(komparatory[i][1], komparatory[i][2]) then
          begin {ak treba začať nový stĺpec komparátorov}
            prazdne_stlpce(stlpec + 1, 3);          {vykresli vodiče}
            for j := 1 to n do plne[j] := false; {inicializuj pole}
            stlpec := stlpec + 3;
            novy_stlpec := false;
          end;
          {vykresli a simuluj komp., zapíš ho pola plne}
          spracuj_komparator(komparatory[i][1], komparatory[i][2], stlpec);
        end;
    end; {for i}
end; { vykresli }

begin       {hlavný program}
  nacitaj;  {načítaj vstup}
  vykresli; {vykresli priebeh výpočtu siete}
  readln;   {počkaj na stlačenie klávesy enter}
  clrscr;   {zmaž obrazovku}
end.
```

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

51. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Leták kategórie P

Vydala IUVENTA
pre vnútornú potrebu Ministerstva školstva SR
Autori príkladov B. Brejová, M. Forišek, M. Pál a T. Vinař
Programom LaTeX sadzbu pripravili T. Vinař a B. Brejová