P-III-1 Pizza vrací úder

Udělejme nejprve několik pozorování: Buď z nejmenší zisk, jehož Marco může dosáhnout. Zřejmě $0 \le z < 2k$, protože pokud by z bylo alespoň $2k$, pak bychom mohli změnit libovolný příjem na výdaj, a tak snížit zisk o nanejvýš $2k$.

Dále nás nezajímá pořadí zadaných čísel – výsledek záleží pouze na tom, kolikrát se které číslo z rozmezí 1 až k vyskytuje. Označme si n_i počet výskytů čísla i. Úlohu si tedy můžeme přeformulovat tak, že hledáme k čísel x_i $(0 \le x_i \le n_i)$ tak, že pokud x_i výskytů čísla i prohlásíme za příjem a $n_i - x_i$ za výdaj, zisk $z = \sum_{i=1}^{k}(2x_i - n_i) \cdot i$ bude nezáporný a nejmenší možný. Označme $(2x_t - n_t) \cdot t$ jako y_t. Výraz y_t může nabýt libovolné z hodnot $-tn_t, -t(n_t-2), \ldots, tn_t$.

Nyní ukážeme, že stačí umět vyřešit úlohu v případě, že všechna čísla n_i mají velikost nanejvýš $2k+1$: Nechť $z = \sum_{i=1}^{k} y_i$ je minimální zisk. Pokud $n_t > 2k+1$ pro nějaké t, snížíme hodnotu n_t o 2 a tvrdíme, že optimální hodnota z pro novou úlohu je stejná jako pro původní úlohu. Stačí tedy dokázat, že původní hodnotu z lze vyjádřit s $x_t < n_t - 2$. Tuto operaci můžeme samozřejmě opakovat, dokud existuje t, pro které je $n_t > 2k+1$ (v programu to realizujeme tak, že pokud $n_t > 2k+1$, rovnou ho nahradíme buď $2k$ nebo $2k+1$ podle toho, zda je n_t sudé či liché).

Podívejme se nyní na vyjádření optimálního zisku $z = \sum_{i=1}^{k} y_i$. Pokud platí $|y_t| < tn_t$, zisk z lze vyjádřit i po snížení n_t. Zbývá vyřešit případ, že $|y_t| = tn_t \ge t(2k+2)$. Předpokládejme, že y_t je kladné, situace, kdy y_t je záporné, se vyřeší analogicky. Jelikož $\sum_{i=1}^{k} y_i < 2k$, musí existovat (nikoliv nutně navzájem různá) čísla $i_1, i_2, \ldots, i_t$ $(1 \le i_j \le k)$ tak, že můžeme postupně zvyšovat y_{i_1} o $2i_1$, y_{i_2} o $2i_2$, …, a takto získané hodnoty budou nekladné.

Uvažme posloupnost součtů 0, i_1, $i_1 + i_2$, …, $i_1 + i_2 + \cdots + i_t$. Mezi těmito $t+1$ součty musí být dva, které dávají stejný zbytek po dělení t, řekněme $i_1 + \cdots + i_p$ a $i_1 + \cdots + i_q$ $(p < q)$, tedy součet $x = i_{p+1} + \cdots + i_q \le kt$ je dělitelný t. Nicméně pak můžeme zmenšit y_t o $2x$ a zvětšit y_i o $2i$ pro všechna $i = i_{p+1}, \ldots, i_q$. Tím se nezmění zisk, ale $|y_t| \le t(n_t - 2)$. Proto lze snížit n_t o 2.

S právě nalezeným pozorováním o omezení na velikost n_t je snadné vytvořit algoritmus založený na dynamickém programování. Pro t od 1 do k si budeme postupně počítat, jakých zisků (či ztrát) lze dosáhnout pouze použitím hodnot y_1, y_2, …, y_t – množinu všech těchto zisků si označme Z_t. Zjevně $Z_0 = \{0\}$ a Z_t vytvoříme ze Z_{t-1} tak, že vezmeme všechna čísla, která jdou vyjádřit jako $x + y$, kde $x \in Z_{t-1}$ a y je jedna z možných hodnot pro y_t. Protože množina Z_t obsahuje čísla s absolutní hodnotou nejvýše $(2k+1)t^2$, velikost každé z množin Z_t je nanejvýš $O(k^3)$. Hledaný zisk z je pak roven nejmenšímu nezápornému číslu, které je obsažené v množině Z_k.

Jelikož možných hodnot pro každé y_t je $O(k)$, jednu množinu Z_t lze snadno zkonstruovat v čase $O(k^4)$. Existuje ale chytřejší algoritmus: číslo z patří do Z_t, pokud alespoň jedno z čísel $z - tn_t, z - t(n_t-2), \ldots, z + tn_t$ patří do množiny Z_{t-1}. My si dokonce pro každé z spočítáme, kolik z těchto čísel patří do Z_{t-1}, označme si tento pročet m_z. Pro prvních $2t$ hodnot z si m_z určíme prostým vyzkoušením všech prvků, což zvládneme v čase $O(k^2)$. Pro každé další z nahlédneme, že m_z je rovno m_{z-2t} zvýšené o jedna, jestliže $z + tn_t \in Z_{t-1}$, a snížené o jedna, jestliže $z - t(n_t+2) \in Z_{t-1}$. Každé další číslo m_z tedy určíme v konstantním čase. Díky omezení na velikost množiny Z_t je časová složitost určení všech hodnot m_z nanejvýš $O(k^3)$.

Čísla n_i lze určit v lineárním čase, celková časová složitost algoritmu tedy je $O(n + k^4)$. Při konstrukci množiny Z_t si stačí pamatovat pouze množinu Z_{t-1}, paměťová složitost proto je $O(k^3)$ (zadaná čísla si nemusíme pamatovat, čísla n_i si můžeme určit již při načítání).

Popsaný algoritmus lze ještě vylepšit, pokud nahlédneme, že stačí uvažovat množiny Z_t obsahující čísla velikosti pouze $O(k^2)$ (tím se časová složitost sníží na $O(n+k^3)$ a paměťová na $O(k^2)$). Toto tvrzení dokážeme podobně jako odhad na velikost čísel n_t. Nechť $z = \sum_{i=1}^{k} y_i$ je minimální zisk, a označme $z_t = \sum_{i=1}^{t} y_i$. Hodnota z_t je číslo, které se pro toto řešení bude vyskytovat v množině Z_t. Dokažme, že lze předpokládat, že $|z_t| \le 2k(k+2)$ pro každé t.

Nechť tomu tak není a $|z_t| > 2k(k+2)$ pro nějaké t. Zabývejme se případem, kdy z_t je kladné, případ, kdy je záporné, se vyřeší analogicky. Pak musí existovat posloupnost čísel c_1, c_2, …, c_x $(1 \le c_j \le t)$ taková, že $\sum_{j=0}^{x} 2c_j = S \ge 2k(k+1)$ a můžeme postupně snižovat y_{c_1} o $2c_1$, y_{c_2} o $2c_2$, …, a takto získané hodnoty budou nezáporné, a posloupnost čísel d_1, d_2, …, d_y $(t < d_j \le k)$ tak, že $0 \le S - \sum_{j=0}^{y} 2d_j < 2k$ a můžeme postupně zvyšovat y_{d_1} o $2d_1$, y_{d_2} o $2d_2$, …, a takto získané hodnoty budou nekladné. Všimněte si, že $x \ge k$ a $y \ge k$. Čísla c_i a $-d_i$ přeuspořádejme do posloupnosti e_1, e_2, …, e_{x+y} takové, že $0 \le \sum_{i=1}^{m} e_i < 2k$ pro každé $m = 0, 1, \ldots, x+y$. Podobně jako v odhadu velikosti n_t nahlédneme, že existují indexy $m_1 < m_2$ takové, že $\sum_{i=m_1}^{m_2} e_i$ je dělitelné $2k$. Protože $|\sum_{i=m_1}^{m_2} e_i| < 2k$, tento součet je roven 0. Nyní provedeme snížení a zvýšení hodnot y_j tak, jak je to naznačeno prvky e_{m_1} e_{m_1+1}, …, e_{m_2}, čímž se jejich absolutní hodnoty zmenší, ale součet se nezmění. Tuto operaci provádíme, dokud existuje t takové, že $|z_t| > 2k(k+2)$. Jelikož vždy zmenšíme absolutní hodnoty některých y_j, po konečném počtu opakování skončíme s řešením, v němž platí $|z_t| \le 2k(k+2)$ pro všechna t.

Program následuje:

```
const MAXK = 100;                       { maximální velikost příjmu či výdaje }
      MAXZ = 2 * MAXK * (MAXK + 2);     { maximální velikost prvku Z_t }

type mnozina = record
       prvky : array[-MAXZ .. MAXZ] of boolean; { true je nastaveno na pozicích všech prvku množiny }
       end;
```

```
var m : array[1 .. 2] of mnozina;       { množiny Z_t a Z_(t+1) }
    predchozi : integer;                { Z_t je m[predchozi], Z_(t+1) je m[3-predchozi] }
    k : integer;
    amaxz : integer;                    { 2k(k+2) }
    ni : array[1 .. MAXK] of integer;   { čísla n_i }
    mz : array[-MAXZ .. MAXZ] of integer; { čísla m_z ... }
                                        { ve skutečnosti by stačilo si pamatovat jen posledních 2t }

{ Inicializuje MN na prázdnou množinu }
procedure smaz (var mn : mnozina);
var i : integer;
begin
  for i := -amaxz to amaxz do
    mn.prvky[i] := false;
end;

{ Vrátí true pokud X je prvkem množiny MN }
function je_prvek (var mn : mnozina; x : integer) : boolean;
begin
  je_prvek := (abs (x) <= amaxz) and mn.prvky[x];
end;

{ Z množiny Z_(T-1) (P) vytvoří množinu Z_T (MN) }
procedure dalsi_zisky (var p, mn : mnozina; t : integer);
var yt, i, z, max_yt : integer;
begin
  max_yt := t * ni[t];

  { spočteme čísla m_z pro z v rozmezí -amaxz ... -amaxz + 2t-1 }
  for z := -amaxz to -amaxz + 2 * t - 1 do
    begin
      i := 0;
      yt := -max_yt;
      while yt <= max_yt do
        begin
          if je_prvek (p, z + yt) then
            inc (i);
          inc (yt, 2 * t);
        end;
      mz[z] := i;
    end;

  { a nyní zbývající hodnoty m_z }
  for z := -amaxz + 2 * t to amaxz do
    begin
      i := mz[z - 2 * t];
      if je_prvek (p, z + max_yt) then
        inc (i);
      if je_prvek (p, z - 2 * t - max_yt) then
        dec (i);
      mz[z] := i;
    end;

  { do množiny mn dáme čísla z taková, že m_z není nula }
  for z := -amaxz to amaxz do
     if mz[z] <> 0 then
        mn.prvky[z] := true;
end;

{ Vrátí nejmenší nezáporné číslo v množině MN }
function minimum (var mn : mnozina) : integer;
var i : integer;
begin
  for i := 0 to amaxz do
    if mn.prvky[i] then
```

```
      begin
        minimum := i;
        break;
      end;
end;

{ Načte seznam příjmů či výdajů }
procedure nacti;
var i, n, a : integer;
begin
  readln (n, k);
  amaxz := 2 * k * (k + 2);

  for i := 1 to k do
    ni[i] := 0;

  for i := 1 to n do
    begin
      read (a);
      inc (ni[a]);
    end;
end;

{ Omezí čísla n_i na nanejvýš 2k+1 }
procedure omez_ni;
var i : integer;
begin
  for i := 1 to k do
    if ni[i] > 2 * k + 1 then
      begin
        if odd(ni[i]) then
          ni[i] := 2 * k + 1
        else
          ni[i] := 2 * k;
      end;
end;

var t : integer;
begin
  nacti;
  omez_ni;
  smaz (m[1]);
  smaz (m[2]);
  predchozi := 1;

  { Z_0 = (0) }
  m[predchozi].prvky[0] := true;

  for t := 1 to k do
    begin
      dalsi_zisky (m[predchozi], m[3 - predchozi], t);
      smaz (m[predchozi]);
      predchozi := 3 - predchozi;
    end;

  writeln (minimum (m[predchozi]));
end.
```

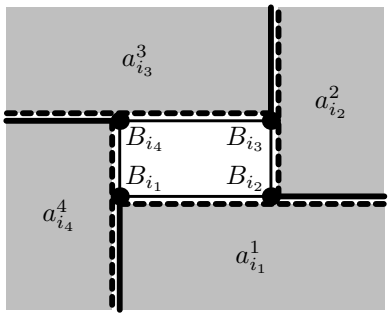
P-III-2 Obdélník

Předvedeme si řešení, jehož časová složitost je $O(N^2)$ a jehož paměťové nároky jsou lineární v počtu daných bodů.

Nejprve vymyslíme, jak pro daný obdélník $A_1A_2A_3A_4$ rychle určit počet bodů, které leží uvnitř tohoto obdélníku. Pro každý bod B_i spočítáme počet bodů $B_{i'}$, jejichž x-ová souřadnice je větší nebo rovna x-ové souřadnici bodu B_i a zároveň y-ová souřadnice je ostře menší než y-ová souřadnice bodu B_i. Tento počet označíme a_i^1. Podobně, a_i^2 bude počet bodů s x-ovou souřadnicí ostře větší a y-ovou souřadnicí větší nebo rovnou, a_i^3 počet bodů s x-ovou souřadnicí menší nebo rovnou

a y-ovou ostře větší a konečně a_i^4 bude počet bodů s x-ovou ostře menší a y-ovou menší nebo rovnou. Každé z čísel a_i^1, a_i^2, a_i^3 a a_i^4 je snadné spočítat v čase $O(N)$ (pro jeden bod B_i). Tedy na spočítání všech $4N$ čísel a_i^1, a_i^2, a_i^3 a a_i^4, $1 \leq i \leq n$, spotřebujeme čas $O(N^2)$.

Je snadné nahlédnout, že pokud vrchol A_j obdélníku $A_1A_2A_3A_4$ je bod B_{i_j}, pak počet bodů ležících uvnitř obdélníku $A_1A_2A_3A_4$ je $N - a_{i_1}^1 - a_{i_2}^2 - a_{i_3}^3 - a_{i_4}^4$ (viz obrázek). Pro jeden obdélník $A_1A_2A_3A_4$ můžeme tedy určit počet bodů, které leží uvnitř tohoto obdélníku, v konstantním čase.



Nyní si popíšeme, jak můžeme rychle najít všechny obdélníky $A_1A_2A_3A_4$, jejichž hrany jsou rovnoběžné s osami a jejichž vrcholy jsou v některých ze zadaných bodů. Nejprve si všechny body setřiďme vzestupně podle x-ové souřadnice a ty body, které mají shodnou x-ovou souřadnici, setřídíme mezi sebou vzestupně podle y-ové souřadnice. Pro každý bod B_i tedy nyní můžeme vypsat ty body, které mají stejnou x-ovou souřadnici jako B_i a větší y-ovou souřadnici, v čase lineárním v jejich počtu. Podobně si setřídíme všechny body podle y-ové souřadnice a v případě shody podle x-ové, abychom mohli snadno nalézt body se stejnou y-ovou a větší x-ovou souřadnicí. Na setřídění bodů potřebujeme čas $O(N \log N)$ (použijeme např. třídící algoritmus heapsort nebo quicksort) a pro uložení setříděného seznamu bodů včetně odkazů do něj pak potřebujeme paměť lineární v N.

Popíšeme si nyní postup, jak nalézt všechny obdélníky $A_1A_2A_3A_4$, které splňují podmínky ze zadání úlohy. Zafixujeme jeden bod B_i a určíme všechny body B_j takové, že bod B_i je vrchol A_1 a bod B_j vrchol A_3 nějakého obdélníku $A_1A_2A_3A_4$ (všimněte si, že vrcholy A_2 a A_4 jsou body B_i a B_j jednoznačně určeny).

K nalezení všech takových bodů B_j pro bod B_i si vytvoříme pomocné pole C, jehož všechny složky $C[j]$ budou na počátku rovny nule. Uvážíme pak všechny body $B_{i'}$, jež mají stejnou x-ovou souřadnici jako B_i a zároveň větší y-ovou souřadnici, a pro každý takový bod $B_{i'}$ uvážíme všechny body $B_{i''}$, které mají stejnou y-ovou a větší x-ovou souřadnici než $B_{i'}$. Body $B_{i'}$ zvládneme najít v čase lineárním v jejich počtu a podobně body $B_{i''}$ pro každý bod $B_{i'}$ lze najít v čase lineárním v jejich počtu. Protože body $B_{i''}$ jsou různé pro různé body $B_{i'}$, trvá nalezení všech bodů $B_{i''}$ čas lineární v N. Složku $C[i'']$ pole C zvýšíme o jedna pro každý bod $B_{i''}$, který jsme takto nalezli. Zdůrazněme, že je nyní každá složka pole C rovna buď 0 nebo 1.

Poté pro bod B_i uvážíme všechny body $B_{i'}$ se stejnou y-ovou a větší x-ovou souřadnicí a pro takové body $B_{i'}$ najdeme všechny body $B_{i''}$ stejnou x-ovou souřadnicí a větší y-ovou souřadnicí. Za každý takový bod zvýšíme složku $C[i'']$ pole C o jedna. Složky pole C jsou nyní rovny 0, 1 nebo 2. Dále platí, že $C[j]$ je rovno 2 tehdy a jen tehdy, jestliže pro body B_i a B_j existuje bod, který má stejnou x-ovou souřadnici jako B_i a stejnou y-ovou souřadnici jako B_j, a zároveň existuje bod, který má stejnou x-ovou souřadnici jako B_j a stejnou y-ovou souřadnici jako B_i. Potom ale takové dva body tvoří vrcholy A_2 a A_4 obdélníku, jehož vrchol A_1 je B_i a vrchol A_3 je B_j. Pro daný bod B_i můžeme tedy v čase $O(N)$ nalézt všechny body B_j, které tvoří protilehlé vrcholy A_1 a A_3 nějakého obdélníku $A_1A_2A_3A_4$. Určit počet vnitřních bodů takovéhoto obdélníku lze pak v konstantním čase, jak jsme si vysvětlili na začátku řešení.

Jak jsme si právě popsali, lze pro daný bod B_i v čase $O(N)$ spočítat počet vnitřních bodů obdélníků $A_1A_2A_3A_4$, jejichž bod A_1 je bod B_i. Takovéto obdélníky spočítáme pro všechny body B_i a vybereme z nich ten, který obsahuje nejvíce vnitřních bodů. Vzhledem k tomu, že bod B_i lze zvolit N způsoby, je časová složitost právě popsaného algoritmu $O(N^2)$. Paměťová složitost je pak lineární v N.

```
program obdelniky;

const MAXN=1000; { maximální počet zadaných bodů; pro jednoduchost používáme statická pole}
var N: word;
    Bx: array[1..MAXN] of integer;
    By: array[1..MAXN] of integer;
    sorted_by_x: array[1..MAXN] of word;
    sorted_by_y: array[1..MAXN] of word;
    index_by_x: array[1..MAXN] of word;
    index_by_y: array[1..MAXN] of word;
    A1, A2, A3, A4: array[1..MAXN] of word;
    C: array[1..MAXN] of byte;
    nejlepsi_pocet: word;
    nejlepsi: array[1..4] of integer;
```

```
procedure nacti;
  var i: word;
  begin
    readln(N);
    for i:=1 to N do readln(Bx[i],By[i]);
  end;

procedure spocitej_A;
  var i,j: word;
  begin
    for i:=1 to N do
      begin
        A1[i]:=0; A2[i]:=0; A3[i]:=0; A4[i]:=0;
        for j:=1 to N do
          begin
            if (Bx[j]>=Bx[i]) and (By[j]<By[i]) then inc(A1[i]);
            if (Bx[j]>Bx[i]) and (By[j]>=By[i]) then inc(A2[i]);
            if (Bx[j]<=Bx[i]) and (By[j]>By[i]) then inc(A3[i]);
            if (Bx[j]<Bx[i]) and (By[j]<=By[i]) then inc(A4[i]);
          end;
      end;
  end;

procedure quick_x(L,P: word);
  var i,j,k:word;
      x,y:integer;
  begin
    i:=L; j:=P;
    x:=Bx[sorted_by_x[(L+P) div 2]];
    y:=By[sorted_by_x[(L+P) div 2]];
    while i<j do
      begin
        while (Bx[sorted_by_x[i]]<x) or
              ((Bx[sorted_by_x[i]]=x) and (By[sorted_by_x[i]]<y)) do
          inc(i);
        while (Bx[sorted_by_x[j]]>x) or
              ((Bx[sorted_by_x[j]]=x) and (By[sorted_by_x[j]]>y)) do
          dec(j);
        if i>j then break;
        k:=sorted_by_x[i]; sorted_by_x[i]:=sorted_by_x[j]; sorted_by_x[j]:=k;
        inc(i); dec(j);
      end;
    if L<j then quick_x(L,j);
    if i<P then quick_x(i,P);
  end;

procedure quick_y(L,P: word);
  var i,j,k:word;
      x,y:integer;
  begin
    i:=L; j:=P;
    x:=Bx[sorted_by_y[(L+P) div 2]];
    y:=By[sorted_by_y[(L+P) div 2]];
    while i<j do
      begin
        while (By[sorted_by_y[i]]<y) or
              ((By[sorted_by_y[i]]=y) and (Bx[sorted_by_y[i]]<x)) do
          inc(i);
        while (By[sorted_by_y[j]]>y) or
              ((By[sorted_by_y[j]]=y) and (Bx[sorted_by_y[j]]>x)) do
          dec(j);
        if i>j then break;
        k:=sorted_by_y[i]; sorted_by_y[i]:=sorted_by_y[j]; sorted_by_y[j]:=k;
        inc(i); dec(j);
      end;
```

```
    if L<j then quick_y(L,j);
    if i<P then quick_y(i,P);
  end;

procedure setrid;
  var i:word;
  begin
    for i:=1 to N do sorted_by_x[i]:=i;
    quick_x(1,N);
    for i:=1 to N do index_by_x[sorted_by_x[i]]:=i;
    for i:=1 to N do sorted_by_y[i]:=i;
    quick_y(1,N);
    for i:=1 to N do index_by_y[sorted_by_y[i]]:=i;
  end;

procedure zkus_vrchol(i: word);
  var i1, i2, j: word;
      a2_index, a4_index: array[1..MAXN] of word;
  begin
    for j:=1 to N do C[j]:=0;
    i1:=index_by_x[i]+1;
    while Bx[sorted_by_x[i1]]=Bx[i] do
      begin
        i2:=index_by_y[sorted_by_x[i1]]+1;
        while By[sorted_by_y[i2]]=By[sorted_by_x[i1]] do
          begin
            inc(C[sorted_by_y[i2]]);
            inc(i2);
            a2_index[i2]:=sorted_by_x[i1];
          end;
        inc(i1)
      end;
    i1:=index_by_y[i]+1;
    while By[sorted_by_y[i1]]=By[i] do
      begin
        i2:=index_by_x[sorted_by_y[i1]]+1;
        while Bx[sorted_by_x[i2]]=Bx[sorted_by_y[i1]] do
          begin
            inc(C[sorted_by_x[i2]]);
            inc(i2);
            a4_index[i2]:=sorted_by_y[i1];
          end;
        inc(i1)
      end;
    for j:=1 to N do
      if (C[j]=2) and (N-A1[i]-A2[a2_index[j]]-A3[j]-A4[a4_index[j]]>nejlepsi_pocet) then
        begin
          nejlepsi_pocet:=N-A1[i]-A2[a2_index[j]]-A3[j]-A4[a4_index[j]];
          nejlepsi[1]:=Bx[i];
          nejlepsi[2]:=Bx[j];
          nejlepsi[3]:=By[i];
          nejlepsi[4]:=By[j];
        end;
  end;

var i: word;

begin
  nacti;
  spocitej_A;
  setrid;
  nejlepsi_pocet:=0;
  for i:=1 to N do zkus_vrchol(i);
  if nejlepsi_pocet=0 then
    writeln('Neexistuje žádný obdélník.')
```

```
  else
    writeln('Obdélník s vrcholy o souřadnicích [', nejlepsi[1], ',', nejlepsi[3],
      '], [', nejlepsi[2], ',', nejlepsi[3], '], [', nejlepsi[2], ',', nejlepsi[4],
      '] a [', nejlepsi[1], ',', nejlepsi[4], '] obsahuje ', nejlepsi_pocet,
      ' vnitřních vrcholů a je to obdélník s nejvíce vnitřními vrcholy.');
end.
```

P-III-3 Grafomat a šamani

Nejprve si rozmyslíme, že stačí umět úlohu vyřešit pro *stromy* (souvislé grafy bez cyklů). Když totiž dostaneme obecný 3-graf, můžeme ho z označeného vrcholu prohledat do šířky a stejně jako v úloze domácího kola si u každého vrcholu zapamatovat, po které hraně jsme do něj přišli. Všimněte si, že tyto hrany tvoří strom s jedním význačným vrcholem, totiž počátečním vrcholem, kterému budeme říkat *kořen.* Navíc tento strom obsahuje všechny vrcholy grafu G, takže je to jeho *kostra.*

> *Důkaz:* Máme ukázat, že graf H tvořený vrcholy grafu G a vybranými hranami je kostrou G. Sledujme běh algoritmu a označme si H_i tu část grafu H, kterou jsme sestrojili během prvních i kroků prohledávání do šířky. V H_i budou tedy právě ty vrcholy, jejichž vzdálenost od kořene je nejvýše i, a hrany, po nichž do těchto vrcholů prohledávání došlo. Graf H_0 obsahuje pouze kořen a až se po k krocích algoritmus zastaví, bude $H_k = H$. Dokážeme indukcí, že žádný graf H_i, a tedy ani H, neobsahuje cyklus. Pro H_0 je to určitě pravda, pokud přecházíme od H_i k H_{i+1}, nemůže žádný nový cyklus vzniknout, jelikož právě přidávané vrcholy ve vzdálenosti $i+1$ připojujeme každý jedinou hranou k nějakému vrcholu z H_i. Zbývá si všimnout, že jelikož G je souvislý, musí každý vrchol grafu G ležet v nějakém H_i, takže i v H.

Chceme tedy rozdělit na poloviny vrcholy zakořeněného stromu, čili obarvit je dvěma barvami tak, aby každou barvu dostala právě polovina vrcholů. Kdybychom nebyli omezeni možnostmi grafomatu, mohli bychom například strom procházet rekurzivně, vždy si pamatovat, jakou barvu jsme naposledy použili, a následující vrchol obarvit barvou opačnou:

> **function** *projdi_strom*(v : vrchol; b : barva) : barva;
> **begin**
> $b := 1 - b$;
> v.barva := b;
> Pro všechny syny w vrcholu v:
> $b :=$ *projdi_strom*(w, b);
> **end;**

Taková funkce projde postupně všechny vrcholy a barvy pravidelně střídá, takže pokud byl počet vrcholů sudý, nutně musí obě barvy použít stejně často.

Na grafomatu můžeme udělat totéž, jen místo rekurze, kterou nemáme k dispozici, použijeme *předávání značek.* Značka vždy ponese informaci o naposledy použité barvě b a bude jednoho z těchto dvou typů: *vstupní* (říká, že jsme vrchol právě navštívili poprvé) nebo *výstupní* (ta signalizuje, že vrchol opouštíme a už se do něj nevrátíme; to odpovídá návratu z funkce *projdi_strom*). Značky budeme předávat tak, aby v každém taktu výpočtu byl označený právě jeden vrchol. Naší procházecí funkci pak budou odpovídat tato pravidla:

1. Na počátku výpočtu obsahuje kořen vstupní značku s barvou 0.
2. Pokud vrchol v obdrží vstupní značku, změní barvu v ní uvedenou na opačnou a obarví se podle ní. Poté:
   a) Pokud má nějaké syny, předá značku prvnímu z nich.
   b) Pokud žádné nemá, značku změní na výstupní a ponechá si ji.
3. Pokud vrchol v obdrží výstupní značku:
   a) Pokud má mladšího bratra (to je vrchol, který má stejného otce a následuje v pořadí synů po v), předá mu vstupní značku se stejnou barvou.
   b) Pokud žádného nemá, předá svou značku svému otci.
   c) Pokud otec neexistuje (tj. výstupní značka doputovala do kořene), skončíme.

Tato pravidla lze přímočaře přepsat do programu pro grafomat. Program poběží v čase lineárním s počtem vrcholů grafu (prohledání do šířky určitě v lineárním čase zvládneme a pak nám bude lineárně dlouho trvat předávání značek, každý vrchol totiž navštívíme dvakrát).

My si ovšem ukážeme o něco efektivnější řešení, které bude lineární vzhledem k *hloubce* stromu, tedy k délce nejdelší cesty z kořene do listu. Bude založené na tom, že strom budeme obarvovat shora dolů po hladinách (vždy všechny vrcholy se stejnou vzdáleností od kořene najednou). Barvy ale závisí i na vrcholech v nižších hladinách, takže si nejdříve předpočítáme, který podstrom má sudou velikost a který lichou (podle toho budeme podstromům říkat *sudé* a *liché*). Algoritmus bude vypadat následovně:

1. Prohledáváme graf z počátečního vrcholu do šířky, čímž vznikne strom.
2. Od listů ke kořeni budeme šířit informace o tom, který podstrom je sudý a který lichý. Listy ihned pošlou, že jsou liché. Ostatní vrcholy počkají, až se rozhodnou jejich synové, a podle toho se pak rozhodnou samy.

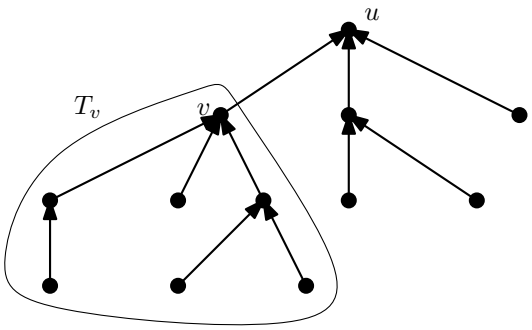
3. Až se rozhodne i kořen (ten bude určitě sudý), začneme po hladinách obarvovat. Nejprve obarvíme kořen barvou 0.
4. Jakmile obarvíme libovolný vrchol v, rozhodneme se, jak obarvit jeho syny. Prvního syna obarvíme opačně než v. Každého dalšího pak buďto stejně jako předchozího nebo opačně podle toho, zda je předchozí podstrom sudý nebo lichý.

Korektnost tohoto algoritmu je zřejmá, jelikož barví stejně jako předchozí algoritmus s předáváním značek, pouze místo čekání na návrat značky rovnou podle sudosti/lichosti podstromu pozná, jaká barva by se vrátila. Časová složitost je lineární vzhledem k hloubce stromu (celkem třikrát procházíme strom po hladinách), v obecném grafu tedy lineární v průměru grafu. Skončíme programem:

```
var x: 0..1;                    { 1=počáteční vrchol, 0=ostatní }
    y: 0..2 = 2;                { výstupní barva: 0=červená, 1=zelená, 2=neurčena }
    z: 0..4 = 0;                { hrana vedoucí k otci vrcholu, 0=neurčena, 4=kořen }
    q: 0..2 = 2;                { parita podstromu: 0=sudá, 1=lichá, 2=neurčena }
    i, k, l: 0..3 = 0;          { pomocné proměnné }

begin
   { 1. fáze: prohledávání do šířky (stavění stromu) }
   if z=0 then
      begin
         { Kořen označíme speciálně }
         if x=1 then z := 4
         { Pokud vrchol ještě nemá otce, vybere si za otce souseda,
           který už otce má, nebo který je počáteční }
         else for i:=1 to 3 do
            if (S[i].z > 0) or (S[i].x = 1) then
               z := i;
      end

   { 2. fáze: počítání parity }
   else if q=2 then
      begin
         l := 0;                { kolik synů má lichou paritu }
         k := 0;                { už můžeme paritu určit? }
         for i:=1 to 3 do
            if S[i].z = 0 then  { neprohledaný soused => určit nemůžeme }
               k := 1
            else if S[i].z = P[i] then           { je to syn }
               if S[i].q = 2 then k := 1         { syn nemá paritu => ani my }
               else if S[i].q = 1 then l := l+1; { lichý syn }
         if k = 0 then
            q := 1 - l mod 2;   { toto funguje i pro listy, ty mají 0 synů }
      end

   { 3. fáze: barvení }
   else if y=2 then
      begin
         if (x=1) and (q<2) then y := 0         { jakmile dopočítáme paritu, kořen obarvíme červeně }
         else if (z>0) and (S[z].y < 2) then    { otec už je obarven => }
            begin                               { => dopočítáme svou barvu podle parity bratrů }
               y := 1 - S[z].y;
               k := P[z];                       { číslo hrany vedoucí z otce sem }
               for i := 1 to k-1 do
                  if S[z].S[i].q = 1 then       { za každého lichého staršího bratra barvu otočíme }
                     y := 1 - y;
            end;
      end

   { Pokud už je vrchol obarven, zastavíme se }
   else stop;
end.
```

P-III-4 Policie zasahuje

V kanalizaci si libovolně zvolíme jedno z větvení, označíme ho u a budeme říkat *ústí*. Ze zadání víme, že kanalizace je taková, že z každého větvení vede právě jedna cesta do ústí. Pro každou stoku je proto jedno z větvení na jejích koncích blíže k ústí než druhé. Stoky si tedy můžeme zorientovat, tj. přiřadit jim směr tak, aby vedly směrem k ústí (viz obrázek). Směr přesně odpovídá tomu, jak by stokou tekly splašky, které bychom pouštěli do všech větvení a které by z kanalizace odtékaly pouze ústím. Pro každé větvení v jsou pak také jednoznačně určena větvení, ze kterých se lze po směru orientace dostat do v (to jsou přesně ta, ze kterých splašky protékají přes v). Množinu těchto větvení včetně větvení v budeme označovat T_v. Všimněte si, že speciálně množina T_u je tvořena všemi větveními v kanalizaci.



Větvením, do kterých jsou napojeny domy mafiánů, budeme říkat *mafiánská*. Pro nalezení potřebného počtu hlídek použijeme jednoduchý rekurzivní algoritmus. Pro každé větvení v postupně spočítáme nejmenší počet hlídek, které je nutné umístit do větvení z množiny T_v, aby každá dvě mafiánská větvení z T_v byla oddělena hlídkou. Navíc ještě určíme, zda existuje rozmístění hlídek v T_v takové, že používá nejmenší možný počet hlídek, a přitom ze žádného mafiánského větvení v T_v neexistuje nehlídaná cesta do v (speciálně, pokud existuje optimální rozmístění hlídek takové, že jedna z hlídek je ve vrcholu v, pak žádná taková cesta neexistuje). Pokud takové (optimální) rozmístění hlídek existuje, budeme říkat, že pro T_v existuje optimální řešení *strážící* v.

Celý výpočet bude probíhat od větvení nejvzdálenějších od ústí kanalizace směrem k ústí kanalizace. Větvení v, která jsou nejdále od ústí kanalizace, jsou zřejmě slepými konci stok. Protože ve slepém větvení není co oddělovat (mafián je v něm nejvýše jeden), není pro hlídání T_v potřeba žádná hlídka. Pokud větvení v není mafiánské, tak vrátíme, že pro T_v existuje optimální řešení strážící v. V opačném případě zjevně nemůže optimální řešení pro T_v být strážící.

Uvažme nyní větvení v, do kterého vedou nějaké stoky v naší orientaci kanalizace. Nechť $v_1, \ldots, v_k$ jsou větvení, ze kterých vede do v stoka orientovaná směrem k v. Nejmenší potřebný počet hlídek pro T_v pak spočteme následovně. Pro každé z větvení $v_1, \ldots, v_k$ jsme si zjistili, kolik je nutné umístit hlídek v částech stoky s větveními z množin $T_{v_1}, \ldots, T_{v_k}$ a dané počty sečteme. Nejmenší počet hlídek potřebný pro T_v je určitě alespoň součet počtů hlídek potřebných pro jednotlivé části $T_{v_1}, \ldots, T_{v_k}$ kanalizační sítě, protože od sebe musíme oddělit mafiánská větvení uvnitř každé z množin $T_{v_1}, \ldots, T_{v_k}$. Nyní ale musíme zajistit, aby byla hlídka i na cestě mezi mafiánskými větveními z různých množin $T_{v_1}, \ldots, T_{v_k}$ – např. mezi mafiánským větvením z T_{v_1} a mafiánským větvením z T_{v_2}, a rozhodnout, zda existuje optimální řešení strážící v. Rozlišíme dva případy:

1. Větvení v je mafiánské. V tomto případě je jasné, že pokud existuje $i \in \{1, \ldots, k\}$ takové, že T_{v_i} nemá optimální řešení strážící v_i, tak je třeba v oddělit od nehlídané množiny, a proto zvýšíme počet hlídek o jedna a novou hlídku umístíme do v. V tomto případě také vrátíme, že T_v má optimální řešení strážící v (zřejmě každá cesta ven z T_v musí projít přes v a v něm je hlídka). Pokud mají všechny množiny $T_{v_1}, \ldots, T_{v_k}$ optimální řešení strážící $v_1, \ldots, v_k$, není třeba žádné mafiány oddělovat. Počet hlídek tedy nemusíme měnit a vrátíme pouze, že množina T_v nemá optimální řešení strážící v – mafiánovi ve v nelze zabránit v odchodu mimo T_v, aniž bychom přidali hlídku do v. Tím bychom ale použili více hlídek, než je nejmenší možný počet hlídek pro oddělení všech mafiánů sídlících v T_v.
2. Větvení v není mafiánské. Pokud alespoň dvě množiny T_{v_i}, T_{v_j}, $i, j \in \{1, \ldots, k\}$, nemají optimální řešení strážící v_i, v_j, musíme do v umístit hlídku, abychom izolovali mafiánská větvení z množin T_{v_i} a T_{v_j}. V tomto případě tedy zvýšíme počet hlídek o jedna a vrátíme, že T_v má optimální řešení strážící v. Pokud existuje nejvýše jedno $i \in \{1, \ldots, k\}$, takové, že T_{v_i} nemá optimální řešení strážící v_i, není třeba nikoho izolovat. Počet hlídek tedy už neměníme a vrátíme, že T_v má optimální řešení strážící v právě když všechny T_{v_i}, $i \in \{1, \ldots, k\}$, mají optimální řešení strážící v_i. I v tomto případě je zřejmé, že pokud vrátíme, že T_v nemá optimální řešení strážící v, tak nelze umístěním nejmenšího možného počtu hlídek zabránit mafiánovi oné T_{v_i}, která nemá optimální řešení strážící v_i, v odchodu skrz v ven.

Počet hlídek vypočtený pro ústí u pak vypíšeme jako výsledek. Už při popisu algoritmu jsme ověřili, že pro každé větvení v skutečně použijeme nejmenší možný počet hlídek potřebný k oddělení mafiánských větvení uvnitř T_v. Počet hlídek spočtený pro u je tedy skutečně nejmenší počet hlídek potřebný k oddělení všech mafiánských větvení.

Algoritmus má lineární časovou i paměťovou složitost.

Nakonec ještě poznamenejme, že se stejnou asymptotickou časovou složitostí lze úlohu řešit i jiným způsobem (ten je ovšem trochu náročnější na implementaci). Je jasné, že pokud na slepý konec stoky není napojen dům mafiána, nehraje tato stoka v našem uvažování roli a můžeme na ní zapomenout. Stejně tak pokud máme větvení, ze kterého vedou dvě stoky a toto větvení není mafiánské, tak můžeme na větvení a přiléhající stoky zapomenout – propojíme však sousední větvení přímou stokou. Pokud nelze provést žádnou z předchozích transformací, je snadné ukázat, že musí existovat větvení, ze kterého vede nejvýše jedna stoka, která není slepá, a ke koncům slepých stok jsou připojeny mafiánské domy. Do takového větvení pak musíme umístit hlídku (buď je slepých stok více a pak musíme mafiány z jejich konců oddělit, nebo je přímo toto větvení mafiánské a pak je třeba oddělit přímo toto větvení od sousedního větvení napojeného na mafiánský dům). Umístěním hlídky ale větvení přestalo hrát v našich úvahách další roli (všechny cesty skrz toto větvení jsou hlídané), proto na něj i na všechny přiléhající stoky můžeme zapomenout a pokračovat opět v transformacích kanalizace. Těmito kroky postupně zmenšujeme kanalizaci, až nám nakonec žádné větvení nezbude. Z našich úvah plyne, že takto zkonstruované řešení úlohy je optimální.

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

#define MAXN 100000

FILE *in, *out;         /* Vstupní a výstupní soubor */
int n, p;               /* Počet větvení a počet mafiánů */
int deg[MAXN];          /* Počet stok vedoucích z větvení */
int e[MAXN][2];         /* Seznam stok */
int edge_ptr[MAXN];     /* Číslo v neib[], kde začínají sousedé každého větvení */
int neib[2*MAXN];       /* Seznamy sousedů větvení */
char podezrely[MAXN];   /* Je do daného větvení připojen dům mafiána? */
int hlidek;             /* Potřebný počet hlídek */

void nacti_vstup(void)
{
  int i, a, b;
  int tmp_ptr[MAXN];

  /* Načteme stoky */
  fscanf(in, "%d %d", &n, &p);
  memset(deg, 0, n*sizeof(int));
  for (i = 0; i < n-1; i++) {
    fscanf(in, "%d %d", &a, &b);
    a--; b--;
    e[i][0] = a;
    e[i][1] = b;
    deg[a]++;
    deg[b]++;
  }

  /* Vytvoříme seznamy sousedů a nastavíme správně jejich počátky */
  edge_ptr[0] = 0;
  for (i = 1; i < n; i++)
    edge_ptr[i] = edge_ptr[i-1]+deg[i-1];
  memset(tmp_ptr, 0, n*sizeof(int));
  for (i = 0; i < n-1; i++) {
    neib[edge_ptr[e[i][0]]+tmp_ptr[e[i][0]]++] = e[i][1];
    neib[edge_ptr[e[i][1]]+tmp_ptr[e[i][1]]++] = e[i][0];
  }

  /* Načteme seznam mafiánů */
  for (i = 0; i < p; i++) {
    fscanf(in, "%d", &a);
    a--;
    podezrely[a] = 1;
  }
}

/* Spočte počet hlídek pro T_v, vrátí 1, pokud je T_v nehlídaná. */
int hledej(int v, int rodic)
```

```
{
  int i, nehlidanych = 0;

  /* Projdeme všechny sousedy, ze kterých k nám vede stoka, a zjistíme situaci */
  for (i = 0; i < deg[v]; i++)
    if (neib[edge_ptr[v]+i] != rodic)
      nehlidanych += hledej(neib[edge_ptr[v]+i], v);
  /* Musíme větvení v hlídat? */
  /* To je třeba buď pokud máme alespoň 2 nehlídané sousedy (pak je třeba
   * izolovat mafiány z nich), nebo pokud ve 'v' sídlí mafián a máme alespoň
   * jednoho nehlídaného souseda. */
  if (nehlidanych > 1 || (podezrely[v] && nehlidanych > 0)) {
      hlidek++;
      return 0;
  }
  /* Jinak nemusíme hlídat, jen vrátíme, jestli je T_v nehlídaný */
  return nehlidanych || podezrely[v];
}

int main(void)
{
  in = fopen("policie.in", "r");
  out = fopen("policie.out", "w");
  nacti_vstup();
  hledej(0,0);
  fprintf(out, "%d\n", hlidek);
  return 0;
}
```

P-III-5 Rybka

Nejjednodušším a zaručeně správným řešením by bylo spočítat si pro každý čas $t = 1, 2, 3, \ldots$ množinu pozic M_t, kde se Julka může nacházet v čase t, z množiny pozic M_{t-1} přidáním všech polí a jejich sousedů do M_t, pokud splňují v časech t i $t-1$ Julčina teplotní omezení. Na začátku v čase $t = 0$ je $M_0 = \{(x_1, y_1)\}$, první M_t ve které bude (x_2, y_2) je první možnost jak se dostat do cíle, pokud bude dřív některá M_t prázdná, Julka se určitě uvaří.

Toto by sice možné velmi snadno naprogramovat, ale časová náročnost by byla až $O(mnT)$, kde T je čas doplutí do cíle nebo přehřátí rybníka – tento může být však velmi velký v porovnání s m a n.

K lepšímu řešení využijeme toho, že stačí u každého pole vědět, kdy nejdříve (a jak) by se na něj mohla Julka dostat. Pokud chci na pole vůbec v průběhu plavby připlout, nemá žádný smysl připlouvat později než při první příležitosti. Na počáteční pole se dostanu v čase 0. Když na nějaké pole už znám nejrychlejší cestu, mohu se z něj na jeho sousedy dostat buď hned nebo musím počkat, až se nové pole ohřeje, nebo vůbec, pokud už je pole moc horké nebo nemůžu čekat dostatečně dlouho.

Pro nalezení nejrychlejší plavby do každého pole lehce upravíme Dijkstrův algoritmus (původně pro hledání nejkratších cest v grafu). Poznamenejme, že detaily důkazu správnosti a popis Dijkstrova algoritmu lze nalézt například na stránkách KSP v sekci Studijní materiály *(http://ksp.mff.cuni.cz/study/cooks.html).* Budeme si udržovat množinu polí Z, kam už známe nejrychlejší cestu, na začátku je $Z = \emptyset$. Dále si pro každé pole budeme udržovat čas $\check{c}(x, y)$, kdy z něj už lze odplout. Na začátku bude $\check{c}(x, y) = \infty$ u všech polí kromě pole (x_1, y_1) a $\check{c}(x_1, y_1) = 0$. Algoritmus v každém kroku přepočítá $\check{c}(x, y)$ sousedů políček ze Z a pak do Z přidá pole s nejmenší hodnotou $\check{c}(x^*, y^*)$, které v Z ještě není. Pokud takto rozšiřujeme Z, nemůžeme se při výpočtu dopustit chyby – kdyby k poli (x^*, y^*) existovala rychlejší plavba než přímo přes pole ze Z, její první pole mimo Z má $\check{c}(x, y)$ určitě nižší než $\check{c}(x^*, y^*)$, což ale není možné. Podobně nahlédneme, že do množiny Z jsou postupně zařazena všechna pole dosažitelná z počátečního pole.

Nyní se zabývejme vlastní implementací tohoto algoritmu. U každého políčka si budeme pamatovat, zda je již v množině Z, současnou hodnotu $\check{c}(x, y)$, počáteční teplotu a směr, odkud je nejlepší na něj přijet. Které pole mimo Z má nejnižší $\check{c}(x, y)$, budeme zjišťovat pomocí haldy (podobně jako v Dijkstrově algoritmu), ve které budou všechna pole neobsažená v množině Z uspořádaná podle $\check{c}(x, y)$. Z haldy v každém kroku algoritmu vybereme pole (x^*, y^*), přidáme ho do Z a podíváme se, jestli by bylo možné doplout do jeho sousedů rychleji (hned nebo s případným čekáním), než jsme zatím uměli. Pokud je to možné, zlepšíme hodnotu $\check{c}(x, y)$ takového souseda, upravíme pozici tohoto pole v haldě a zapamatujeme si nový lepší směr připlutí, abychom později byli schopni pro každé pole v Z vystopovat optimální cestu až do počátku. Pokud bychom měli z haldy vybrat pole s $\check{c}(x, y) = \infty$, skončíme. Pokud se ale dřív pole (x_2, y_2) dostane do množiny Z, nalezli jsme nejrychlejší možný způsob plavby na toto pole a také skončíme.

Složitost algoritmu při každém z nanejvýš mn rozšíření množiny Z je $O(\log(mn))$ – potřebujeme z haldy vybrat minimum a aktualizovat až 4 sousední hodnoty a každá operace s haldou trvá logaritmicky s její velikostí. Načtení dat ze vstupu, příprava haldy i vypsání výsledku netrvá déle než $O(mn)$. Celková časová složitost popsaného algoritmu je tedy $O(mn \log(mn))$ a paměťová složitost $O(mn)$.

```
program rybka;

const   MAXM=1000;
        MAXN=1000;
        INFTY=3000000;

type    PPole=^TPole;
        TPole=record
                x,y:longint;
                halda:longint;
                t:longint;
                c:longint;
                zpole:PPole;
        end;

var     m,n,x1,y1,x2,y2,t1,t2:longint;          { zadané parametry }
        rybnik:array[1..MAXM,1..MAXN] of TPole; { pole rybníka }
        halda:array[1..MAXM*MAXN] of PPole;     { halda v poli }
        velhaldy:longint;

procedure prehod(pos1:longint; pos2:longint);   { přehoď v haldě dvě pole }
var tmp:PPole;
begin
        halda[pos1]^.halda:=pos2;
        halda[pos2]^.halda:=pos1;
        tmp:=halda[pos1];
        halda[pos1]:=halda[pos2];
        halda[pos2]:=tmp;
end;

procedure upravhaldu(var pole:TPole);           { sniž hodnotu c(x,y) pole v haldě }
var     hpos:longint;
begin
        hpos:=pole.halda;
        while (hpos>1) and (halda[hpos]^.c < halda[hpos div 2]^.c) do begin
                prehod(hpos,hpos div 2);
                hpos:=hpos div 2;
        end;
end;

function vyberzhaldy:PPole;                     { vyber z haldy pole s nejnižším c(x,y) }
var     hpos,minpos:longint;
        hotovo:boolean;
begin
        if velhaldy=0 then
                vyberzhaldy:=nil
        else begin
                vyberzhaldy:=halda[1];
                halda[1]:=halda[velhaldy];
                halda[1]^.halda:=1;
                dec(velhaldy);
                hotovo:=false;
                hpos:=1;
                repeat
                        { buď už jsme na spodku haldy }
                        if (hpos*2>velhaldy) then hotovo:=true
                        { nebo máme jen jednoho potomka }
                        else if (hpos*2=velhaldy) then begin
                                hotovo:=true;
                                if (halda[hpos]^.c>halda[hpos*2]^.c) then
                                        prehod(hpos,hpos*2);
                        end
                        { nebo máme dva potomky }
                        else begin
                                minpos:=hpos;
```

```
                                if (halda[minpos]^.c>halda[hpos*2]^.c) then
                                        minpos:=hpos*2;
                                if (halda[minpos]^.c>halda[hpos*2+1]^.c) then
                                        minpos:=hpos*2+1;
                                if (minpos=hpos) then
                                        hotovo:=true
                                else begin
                                        prehod(hpos,minpos);
                                        hpos:=minpos;
                                end;
                        end;
                until hotovo;
        end;
end;

procedure vypis(var f:text; var pole:TPole);
begin
        if pole.zpole<>nil then begin
                vypis(f,pole.zpole^);
                if pole.c-pole.zpole^.c>1 then write(f,pole.c-pole.zpole^.c-1,' ');
                if pole.x>pole.zpole^.x then write(f,'V ');
                if pole.x<pole.zpole^.x then write(f,'Z ');
                if pole.y>pole.zpole^.y then write(f,'J ');
                if pole.y<pole.zpole^.y then write(f,'S ');
        end;
end;

procedure vylepsipole(var ktere:TPole; var odkud:TPole);
var cek:longint;
begin
        { jak dlouho musím počkat, než se pole dost ohřeje? }
        cek:=t1-(odkud.c+ktere.t);
        if cek<0 then cek:=0;
        { jedná se o cílové pole? pokud ano, vůbec nečekej }
        if (ktere.x=x2) and (ktere.y=y2) then begin
                ktere.c:=odkud.c+1;
                ktere.zpole:=@odkud;
                upravhaldu(ktere);
        end
        { když počkám tolik, bude to ok na tomto i cílovém poli? je to lepší cesta?}
        else if (odkud.c+cek+odkud.t<=t2) and (odkud.c+cek+ktere.t+1<=t2)
             and (ktere.c>odkud.c+cek+1) then begin
                ktere.c:=odkud.c+cek+1;
                ktere.zpole:=@odkud;
                upravhaldu(ktere);
        end;
end;

var     min:PPole;
        hotovo:boolean;
        f1,f2:text;
        i,j:longint;
begin
        assign(f1,'rybka.in');
        reset(f1);
        assign(f2,'rybka.out');
        rewrite(f2);
        readln(f1,m,n,t1,t2);
        readln(f1,x1,y1,x2,y2);
        for j:=1 to n do
                for i:=1 to m do begin
                        read(f1,rybnik[i,j].t);
                        rybnik[i,j].x:=i;
                        rybnik[i,j].y:=j;
                        rybnik[i,j].c:=INFTY;
```

```
                        rybnik[i,j].zpole:=nil;
                        rybnik[i,j].halda:=velhaldy+1;
                        halda[velhaldy+1]:=@rybnik[i,j];
                        inc(velhaldy);
                end;
        rybnik[x1,y1].c:=0;
        upravhaldu(rybnik[x1,y1]);

        { hlavní smyčka výběru z haldy }
        hotovo:=false;
        repeat
                min:=vyberzhaldy;
                { buď už v haldě nic k přidání není }
                if (min=nil) or (min^.c=INFTY) then begin
                        writeln(f2,'Chudak Julka!');
                        hotovo:=true;
                end
                { nebo vybírám cílové pole }
                else if (min^.x=x2) and (min^.y=y2) then begin
                        hotovo:=true;
                        vypis(f2,rybnik[x2,y2]);
                        { Ať žije Julka! }
                end
                { nebo je to nějaké obecné pole }
                else begin
                        if min^.x>1 then vylepsipole(rybnik[min^.x-1,min^.y],min^);
                        if min^.y>1 then vylepsipole(rybnik[min^.x,min^.y-1],min^);
                        if min^.x<M then vylepsipole(rybnik[min^.x+1,min^.y],min^);
                        if min^.y<N then vylepsipole(rybnik[min^.x,min^.y+1],min^);
                end;
        until hotovo;
        close(f1);
        close(f2);
end.
```