MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

53. ročník, školský rok 2003/2004

Riešenia úloh III. kola kategórie P

1. súťažný deň

P-III-1

Úlohu reprezentujeme pomocou orientovaného grafu. Agenti predstavujú vrcholy grafu. Skutočnosť, že agent a môže vydať rozkaz agentovi b, vyjadríme pomocou orientovanej hrany (a, b). Našou úlohou je nájsť v tomto grafe vrchol, z ktorého sa môžeme dostať do každého iného vrcholu.

Uvažujme nasledujúci algoritmus. Začneme prehľadávaním do hĺbky z ľubovoľného vrcholu grafu. Ak prehľadávaním prejdeme všetky vrcholy, našli sme šéfa; v opačnom prípade pokračujeme tým, že začneme nové prehľadávanie do hĺbky v jednom z vrcholov, ktorý sme ešte neprehľadali (doteraz prehľadané vrcholy pritom necháme označené ako prehľadané). To opakujeme, až kým neprehľadáme všetky vrcholy nášho grafu. Nech r je vrchol, ktorý je začiatkom posledného prehľadávania.

Tvrdenie

Ak náš graf má (aspoň jedného) šéfa, tak vrchol r je šéfom.

Dôkaz

Predpokladajme, že náš graf má šéfa a že vrchol r nie je šéfom. Nech je šéfom vrchol s. Musíme uvažovať dve možnosti:

- **Vrchol s bol objavený v poslednom prehľadávaní.** To by ale znamenalo, že sa do tohoto vrcholu môžeme dostať z vrcholu r (lebo vrchol r je počiatkom tohto prehľadávania) a teda sa môžeme dostať z vrcholu r do ľubovoľného iného vrcholu cez s, čo je ale v spore s našim predpokladom, že r nie je šéfom.
- **Vrchol s bol objavený skôr ako v poslednom prehľadávaní.** Keďže sa však z vrcholu s dá dostať do ľubovoľného vrcholu, museli by sme potom vrchol r objaviť v tom istom prehľadávaní ako s, a teda vrchol r nemôže byť začiatkom posledného prehľadávania.

Zostáva nám teda jedine overiť (opäť prehľadávaním do hĺbky), či r je skutočne šéfom grafu; v opačnom prípade graf nemá šéfa. Časová zložitosť celého algoritmu je $O(M + N)$, kde N je počet vrcholov a M je počet hrán grafu.

Iný, na implementáciu zložitejší algoritmus: Nájdeme silno súvislé komponenty grafu na vstupe. Topologicky maximálny je každý komponent, do ktorého nevedie žiadna hrana. Ak je takýchto komponentov viac, graf nemá šéfa. Ak je jeden, šéfmi sú práve vrcholy v ňom.

program *agenti*;

const $MAXM = 10000$;

```
    MAXN = 100;

var rozkazuje     : array [1..MAXM] of integer;
    ind_od, ind_do : array [1..MAXN] of integer;
    { agent i+6 rozkazuje agentom
      rozkazuje[ind_od[i]]...rozkazuje[ind_do[i]] }
    N             : integer;
    prehladany    : array [1..MAXN] of boolean;

procedure nacitaj;
var i, M, agent : integer;

begin
    write('Pocet agentov:'); readln(N);

    M := 0;
    for i := 1 to N do begin
        write('Agent ', i + 6, ' rozkazuje: (ukonci -1)');
        ind_od[i] := M + 1;
        read(agent);
        while (agent > 0) do begin
            M := M + 1;
            rozkazuje[M] := agent - 6;
            read(agent);
        end;
        ind_do[i] := M;
    end;
end; { nacitaj }

procedure prehladaj(i : integer);
var j             : integer;

begin
    if not prehladany[i] then begin
        prehladany[i] := true;
        for j := ind_od[i] to ind_do[i] do begin
            prehladaj(rozkazuje[j]);
        end;
    end;
end; { prehladaj }

var i, posledny : integer;
    je_sef      : boolean;

begin
    nacitaj;

    { nainicializuj pole prehladanych vrcholov }
    for i := 1 to N do prehladany[i] := false;
```

```
    { prehladavaj, az kym neostanu ziadne vrcholy }
    for i := 1 to N do begin
      if not prehladany[i] then begin
        posledny := i;
        prehladaj(i);
      end;
    end;

    { znovu nainicializu pole prehladanych vrcholov }
    for i := 1 to N do prehladany[i] := false;
    prehladaj(posledny);

    { zisti, ci sme prehladali vsetky vrcholy }
    je_sef := true;
    for i := 1 to N do je_sef := je_sef and prehladany[i];

    if je_sef then
      writeln('Sefom je agent ', posledny + 6)
    else
      writeln('Nie je ziadny sef');
end. { agenti }
```

P-III-2

Ľahko zostrojíme riešenie, ktoré potrebuje čas $O(K)$ na spracovanie jednej hodnoty na vstupe. Stačí si v cyklicky prepisovanom poli pamätať posledných K hodnôt. Zakaždým, keď prečítame ďalšie číslo zo vstupu, pole prejdeme a vypíšeme najmenšiu z hodnôt v ňom.

Ukážeme lepšie riešenie, ktoré bude potrebovať na spracovanie jedného čísla čas $O(\log K)$. Predstavme si, že by sme si aktuálnych K hodnôt udržiavali v halde. Novú hodnotu do tejto haldy ľahko pridáme v čase $O(\log K)$. Pred tým, ako vypíšeme minimum (ktoré je v koreni haldy), potrebujeme z haldy vyhodiť najstaršiu hodnotu. Odkiaľ ale máme vedieť, ktorá z nich to je?

Pomôžeme si tak, že hodnoty, ktoré nám budú prichádzať, vložíme nielen do haldy, ale aj do fronty. Medzi týmito dátovými štruktúrami si budeme udržiavať smerníky, aby sme v každom okamihu vedeli o každom prvku fronty povedať, kde je v halde a naopak.

Keď teda príde nová hodnota, vložíme ju do haldy a na koniec fronty. Následne zo začiatku fronty vyberieme najstaršiu hodnotu, pomocou smerníka ju nájdeme v halde a odstránime ju aj odtiaľ. Teraz už len vypíšeme hodnotu v koreni haldy.

Obe operácie s haldou majú časovú zložitosť $O(\log K)$, zvyšné operácie vieme spraviť v konštantnom čase. Pamäť spotrebovaná haldou aj frontou je $O(K)$.

Existuje však ešte lepšie riešenie. Stačí si uvedomiť, že ak nám príde teplota T, môžeme zabudnúť všetky skoršie teploty, ktoré sú od T väčšie alebo rovné – zjavne žiadna z nich už nikdy nebude najmenšia. Teploty, ktoré si ešte pamätáme, budeme udržiavať v poradí, v akom boli na vstupe. Uvedomte si, že potom budú v každom okamihu pamätané teploty tvoriť rastúcu postupnosť.

Čo sa stane, keď príde nová teplota? Niekoľko najväčších pamätaných teplôt si môžeme

prestať pamätať. To spravíme ľahko, lebo tieto teploty sú práve na konci zoznamu. Na koniec zoznamu zaradíme práve prečítanú teplotu. Na začiatku zoznamu je najstaršia pamätaná teplota, ak už je pristará, vyhodíme ju. V tomto okamihu je na začiatku zoznamu najmenšia teplota z posledných K prečítaných. Vypíšeme ju a pokračujeme ďalej.

Spracovanie jednej teploty má pri tomto riešení *amortizovanú* časovú zložitosť $O(1)$. To znamená, že síce spracovanie jednej konkrétnej teploty môže trvať dlho (ak vyhadzujeme veľa starších teplôt), ale na spracovanie N teplôt nám bude určite stačiť čas $O(N)$. To preto, že každú teplotu do zoznamu raz vložíme a raz z neho vyhodíme. (Predchádzajúce riešenie potrebovalo na spracovanie N teplôt čas $O(N \log K)$.) Pamäťová zložitosť je opäť $O(K)$.

```
#include <iostream>
#include <deque>
using namespace std;

typedef struct { double T; int when; } teplota;
deque<teplota> D;

int main(void){
  int K;
  teplota tmp;

  cin >> K;
  D.clear(); tmp.when=0;
  while (1) {
    cin >> tmp.T; tmp.when++;
    if (tmp.T==-1000) break;
    while (!D.empty()) if (D.back().T >= tmp.T) D.pop_back(); else break;
    if (!D.empty()) if (D.front().when==tmp.when-K) D.pop_front();
    D.push_back(tmp);
    cout << D.front().T << "\n";
  }
  return 0;
}
```

P-III-3

a) Na úvod si pripomeňme, že v riešeniach krajského kola ste sa mohli dočítať okrem iného o tom, ako pomocou dvoch počítadiel simulovať zásobník. Pre istotu zopakujeme, ako na to:

Zásobník si vieme simulovať v jednom registri (za pomoci druhého). Odteraz budú písmená a, b, c zodpovedať číslam 1, 2, 3. Číslo v registri R_1 bude predstavovať náš zásobník – keď ho zapíšeme v sústave so základom 4, jednotlivé cifry budú predstavovať vložené hodnoty (cifra na mieste jednotiek bude naposledy vložená hodnota). Teda keď do prázdneho zásobníka uložíme postupne písmená a, c, b, a, bude v R_1 hodnota $a\times4^3+c\times4^2+b\times4+a = 1\times4^3+3\times4^2+2\times4+1 = 64+48+8+1 = 121$.

Ako ale s takýmto registrom-zásobníkom pracovať? Vložiť novú hodnotu x je jednoduché – za pomoci registra R_2 vynásobíme obsah R_1 štyrmi a potom ho x-krát zväčšíme o 1. Takisto vybrať naposledy vloženú hodnotu nie je ťažké – je to presne opačná operácia. Vydelíme obsah

registra R_1 štyrmi. Zvyšok po delení je naposledy vložená hodnota, podiel (ktorý dostaneme v R_2) je zásobník bez tejto hodnoty.

Ako ale vyriešiť zadanú úlohu? Jedna možnosť je simulovať (pomocou dvoch zásobníkov) frontu, v ktorej si udržujeme tie písmená, ktorých pár sme ešte nevideli. Toto riešenie je pomerne komplikované, jeho základná myšlienka je, že prichádzajúce písmená vkladáme do prvého zásobníka, písmená na kontrolu vyberáme z druhého zásobníka a vždy, keď sa nám druhý zásobník vyprázdni, doň presypeme obsah prvého zásobníka.

Ukážeme si jednoduchšie riešenie. Budeme opäť používať dva zásobníky. Do prvého budeme vkladať všetky prichádzajúce malé písmená (ako hodnoty 1,2,3), do druhého veľké (tiež ako hodnoty 1,2,3). Nuž a po dočítaní slova sa jednoducho zahľadíme na oba zásobníky. Vstupné slovo bolo dobré práve vtedy, ak ich obsah je rovnaký. To ľahko overíme.

```
var vstup : char;
    i, co : byte;
begin
   Read(vstup);
   while (vstup <> '$') do begin
      if (vstup >= 'a') then begin
         if (vstup = 'a') then co := 1;
         if (vstup = 'b') then co := 2;
         if (vstup = 'c') then co := 3;

         while not Zero(R_1) do begin Dec(R_1); for i := 1 to 4 do Inc(R_0); end;
         while not Zero(R_0) do begin Dec(R_0); Inc(R_1); end;
         while (co > 0) do begin Inc(R_1); co := co - 1; end;
      end else begin
         if (vstup = 'A') then co := 1;
         if (vstup = 'B') then co := 2;
         if (vstup = 'C') then co := 3;

         while not Zero(R_2) do begin Dec(R_2); for i := 1 to 4 do Inc(R_0); end;
         while not Zero(R_0) do begin Dec(R_0); Inc(R_2); end;
         while (co > 0) do begin Inc(R_2); co := co - 1; end;
      end;
      Read(vstup);
   end;
   while (not Zero(R_1)) and (not Zero(R_2)) do begin Dec(R_1); Dec(R_2); end;
   if Zero(R_1) and Zero(R_2) then Accept;
end.
```

b) Kvôli prehľadnosti nech pôvodné registre sú R_1, R_2, R_3 a nové registre Q_1, Q_2.

Použijeme myšlienku podobnú tej z domáceho kola. Zakódujeme obsah všetkých troch registrov do jedného, druhý register budeme používať ako pomocný pri práci s prvým. Takže namiesto troch registrov s obsahom a, b, c budeme mať jeden register Q_1 s obsahom $2^a 3^b 5^c$. Pri simulovaní každej z operácií použijeme Q_2 ako pomocný register. Ako na začiatku, tak aj po odsimulovaní každej z operácií v ňom bude uložená nula.

Operácie **Inc**(R_x) v pôvodnom programe nahradíme tým, že obsah nového registra Q_1 vynásobíme 2, 3, resp. 5. Podobne príkaz **Dec**(R_x) nahradíme príslušným delením.

Odsimulovať podmienku **Zero**(R_x) bude trošku komplikovanejšie. Počas vyhodnocovania nejakej zloženej podmienky totiž nemôžeme robiť operácie s registrami – zistiť, či je v R_x nula

teda musíme **pred** vyhodnotením príslušnej podmienky. Navyše drobné problémy spôsobí, že táto podmienka sa môže vyskytovať aj v podmienke pre príkaz **while**, kde ju treba vyhodnocovať pri každej iterácii (a nie len pred prvým volaním **while**).

Definujme „makro“ (kus výpočtu) **SpocitajZ**(R_x), ktoré bude fungovať nasledovne: Ak je v R_x nula, po jeho vykonaní bude v premennej z kladná hodnota, inak tam bude 0. Toto makro bude fungovať nasledovne: Začneme tým, že obsah Q_1 vydelíme príslušným prvočíslom, pričom si (v premennej z) zapamätáme zvyšok, ktorý sme dostali po tomto delení. Vrátime obsah Q_1 do pôvodného stavu. Ak obsah Q_1 bol deliteľný príslušným prvočíslom (a teda neplatí **Zero**($\mathbf{R_x}$)), bude v z nula, inak tam bude kladný zvyšok. Výraz **Zero**(R_x) má teda v tomto okamihu rovnakú pravdivostnú hodnotu ako výraz $(z > 0)$.

Každý príkaz „**if** P **then** príkazy;“ nahradíme makrom **SpocitajZ**(R_x) a príkazom „**if** P' **then** príkazy;“, kde podmienka P' vznikla z P tak, že sme namiesto všetkých výskytov výrazu **Zero**(R_x) dali výraz $(z > 0)$.

Každý príkaz „**while** P **do** príkazy;“ nahradíme nasledujúcim kusom výpočtu: „**SpocitajZ**(R_x); **while** P' **do begin** príkazy; **SpocitajZ**(R_x); **end**;“

Nové „makrá“ **Inc**, **Dec** (ktorými nahradíme každý výskyt týchto príkazov v pôvodnom programe) a **SpocitajZ** (na simuláciu **Zero**) budú teda vyzerať nasledovne:

```
var x, y, z, i : byte; { nove premenne, neboli v povodnom programe }

{ Inc(R_x) – x je v premennej x, predpokladame, ze Zero(Q_2) }

  if (x = 1) then y := 2 else if (x = 2) then y := 3 else y := 5;
  { vynasobime obsah Q_1 cislom y }
  while (not Zero(Q_1)) do begin
     Dec(Q_1);
     for i := 1 to y do Inc(Q_2);
  end;
  while (not Zero(Q_2)) do begin
     Dec(Q_2); Inc(Q_1);
  end;

{ Dec(R_x) – x je v premennej x, predpokladame, ze Zero(Q_2) }

  if (x = 1) then y := 2 else if (x = 2) then y := 3 else y := 5;
  z := 0;
  { vydelime obsah Q_1 cislom y }
  while (not Zero(Q_1)) do begin
     Dec(Q_1);
     if (Zero(Q_1)) then begin z := 1; break; end; { v R_x bola nula }
     for i := 1 to y - 1 do Dec(Q_1);
     Inc(Q_2);
  end;
  if (z = 1) then begin
     { obnovime povodny stav Q_1 – nic sa nemeni }
     while (not Zero(Q_2)) do begin
        Dec(Q_2); for i := 1 to y do Inc(Q_1);
     end;
     Inc(Q_1);
```

```
  end else begin
     { presunieme do Q_1 podiel }
     while (not Zero(Q_2)) do begin
        Dec(Q_2); Inc(Q_1);
     end;
  end;

{ SpocitajZ(R_x) – x je v premennej x, predpokladame, ze Zero(Q_2) }

  if (x = 1) then y := 2 else if (x = 2) then y := 3 else y := 5;
  { vydelime obsah Q_1 cislom y }
  z := 0;
  while (not Zero(Q_1)) do begin
     Dec(Q_1);
     z := z + 1;
     if (z = y) then begin z := 0; Inc(Q_2); end;
  end;
  { vratime spaet povodnu hodnotu do Q_1 }
  while (not Zero(Q_2)) do begin
     Dec(Q_2); for i := 1 to y do Inc(Q_1);
  end;
  for i := 1 to z do Inc(Q_1);
  { a v premennej z mame hladany zvysok }
```

Dva dôležité detaily, ktoré ste si mohli všimnúť: 1. Netreba zabudnúť ošetriť situáciu, ak register R_i obsahuje nulu. V tomto prípade aj po vykonaní **Dec**(R_i) musí v R_i (podľa definície) zostať nula.

2. Ak sme pri simulovaní príkazov **Inc**, **Dec** a **Zero** potrebovali použiť premenné, muselo ísť o **nové** premenné, ktoré sa dovtedy v programe nevyskytovali. (Čo ak napr. pôvodný program obsahoval časť `for i:=1 to 3 do Inc(R_1)` a my pri simulácií **Inc**(R_1) použijeme i?) Voľných mien pre nové premenné máme k dispozícii nekonečne veľa, ľahko nájdeme nejaké nepoužité.

Ľahko nahliadneme, že keď takto upravíme ľubovoľný program, tak upravený program bude ekvivalentný s pôvodným – t.j. dá na každom vstupe rovnaký výstup ako pôvodný program. Nuž a ak pôvodný program používal tri registre, upravený program už používa len dva.

Uvedomte si, že použitím tohto postupu na program používajúci $k > 3$ registrov dostaneme program, používajúci $k - 1$ registrov. Preto platí pomerne prekvapivý výsledok: K ľubovoľnej úlohe, ktorú vieme riešiť na registrovom počítači, existuje program, ktorému na jej riešenie stačia dva registre.

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

53. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia III. kola kategórie P

1. súťažný deň

Vydala IUVENTA
pre vnútornú potrebu Ministerstva školstva SR
Sadzba programom LaTeX
Zodpovedný redaktor M. Forišek

MATEMATICKÁ OLYMPIÁDA
NA STREDNÝCH ŠKOLÁCH

53. ročník, školský rok 2003/2004

Riešenia úloh III. kola kategórie P

2. súťažný deň

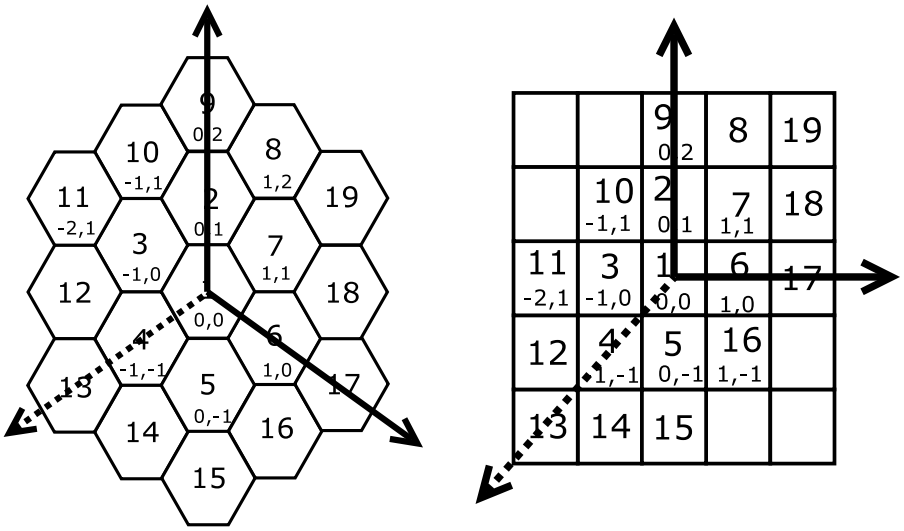
P-III-4

Na poskakovanie psíkov sa môžeme pozerať ako na hru. Stav hry sa dá jednoznačne popísať pozíciou obidvoch psíkov. Skok psíkov predstavuje ťah. Keď psíci skočia, zmenia stav hry. Povolené stavy hry budú tie, ktoré zodpovedajú povoleným pozíciám psíkov. Pre každý stav hry budeme skúmať, na koľko ťahov sa dá do neho dostať zo začiatočného stavu (keď obidvaja psíci sú na začiatočných miestach). Tento počet ťahov budeme označovať vzdialenosťou daného stavu.

Vzdialenosť začiatočného stavu je 0. Všetky stavy, do ktorých sa dá z neho dostať na jeden ťah budú vo vzdialenosti 1. Teraz prejdeme cez všetky stavy vo vzdialenosti 1 a hľadáme, do ktorých nových stavov sa z nich vieme dostať – zjavne budú vo vzdialenosti 2. Takto môžeme analogicky pokračovať pre stavy vo vzdialenosti 3, 4, ... Skončíme, keď nájdeme koncový stav (obidvaja psíci sú na koncových miestach), alebo sme už neobjavili žiadny nový stav. Táto technika prehľadávania stavov sa nazýva prehľadávanie do šírky.

Otázkou ostáva, ako pre každý stav určiť, do ktorých ďalších (susedných) stavov za z neho dá dostať na jeden ťah. Dosť by nám pomohlo, keby sme vedeli pre každé políčko na lúke povedať čísla jeho susedov. Susedia stavu by sa potom našli ľahko. Zo všetkých možných pohybov obidvoma psíkmi 6 smermi (36 možností) vyškrtáme skákanie rovnakým smerom, skákanie po bodliakoch, skočenie mimo lúky a skočenie na to isté políčko.

Aby sa nám hľadali susedia ľahšie, ukážeme si, ako sa dá 6-uholníkový plánik lúky reprezentovať v obyčajnom dvojrozmernom poli. Na políčku 1 si vyberieme dva smery. Jeden ukazuje stúpajuci smer prvej súradnice, druhý smer druhej súradnice. Takto sme priradili ku každému políčku súradnice x, y, čomu zodpovedajú indexy v obyčajnom dvojrozmernom poli. V dvojrozmernom poli nájsť susedov je už ľahké. Konkrétne pri našej voľbe súradnicových osí budú susedia políčka x, y na políčkach $x, y+1$, $x-1, y$, $x-1, y-1$, $x, y-1$, $x+1, y$ a $x+1, y+1$.



Ako zistíme pre políčko s číslom k jeho súradnice? Všimnime si, že špirálu môžeme rozložiť na vrstvy šesťuholníkového tvaru. Najprv si zistíme, na koľkej vrstve špirály sa k nachádza, potom stranu na vrstve, pozíciu na strane a je to.

Počet políčok na 0-tej vrstve špirály: 1

Počet políčok na i-tej vrstve špirály: Vrstva špirály má 6 strán, na každej je i políčok – $6i$

Počet políčok na špirálach $0..v$: $1 + \sum_{i=1}^{v} 6i = 3v^2 - 3v + 1$

Vrstva špirály, kde sa nachádza políčko k: Hľadáme kladné riešenie rovnice $k = 3v^2 - 3v + 1$ a zaokrúhlime ho nahor. Po vyriešení dostaneme $v = \left\lceil \frac{1}{2} + \sqrt{\frac{k}{3} - \frac{1}{12}} \right\rceil$. (Jednoduchší a trochu pomalší postup: zvyšujeme v, kým počet políčok nedosiahne k.)

Strana na špirále, kde sa nachádza políčko k: Poznáme už vrstvu v, kde sa k nachádza. Odrátame od k celkový počet políčok na skorších vrstvách a ešte 1. Takto dostaneme poradové číslo políčka v tejto vrstve (číslované od 0). Na vrstve v má každá strana v políčok, rozdiel teda stačí predeliť v. Číslo strany s bude podiel, pozícia na strane p bude zvyšok po tomto delení.

Už vieme pre dané políčko k zrátať jeho vrstvu v, stranu s a pozíciu na strane p. Z týchto hodnôt dostaneme súradnice podľa nasledujúcej tabuľky:

s	x	y
0	+v-1-p	+v
1	-1-p	+v-1-p
2	-v	-1-p
3	-v+1+p	-v
4	+1+p	-v+1+p
5	+v	+1+p

Implementácia Stav hry je jednoznačne repezentovaný súradnicami x_1, y_1, x_2, y_2 obidvoch psíkov. Pri prehľadávaní do šírky si pamätáme zoznam stavov, ktoré sme už videli, ale ešte sme z nich neskúšali prehľadávať nové vrcholy. Na to nám poslúži fronta. Ďalej potrebujeme vedieť pre každý stav rýchlo zistiť, či sme ho ešte nevideli, už videli, alebo sa do neho nedá ísť (bodliaky). Na to máme vyhradené štvorrozmerné pole, kde je to priamo zapísané. (Presne toto isté sa dalo spraviť dvojrozmerným poľom, do ktorého by sme indexovali pôvodnými súradnicami. Tam by sme ale navyše potrebovali vedieť zo súradníc povedať pôvodné číslo políčka.)

Aby sme nemuseli pri prehľadávaní do šírky kontrolovať, či nevylezieme von z lúky, postavíme okolo celej lúky bodliaky. Rovnako zakážeme stavy, kde by boli obidvaja psíci na jednom mieste.

Prehľadávame priestor o veľkosti rádovo $O(N^2)$, kde N je veľkosť lúky. Časová aj pamäťová zložitosť prehľadávania je lineárna od veľkosti tohto priestoru, teda $O(N^2)$.

```
program psici;
const
  EPS = 1.0E-6;   {max. chyba vzniknuta v realnych cislach}
  MAX_V = 16;     {najvacsia mozna vrstva (aj so zarazkou)}
  MAX_STAVOV = MAX_V*MAX_V*MAX_V*MAX_V;
  INF = 299999;   {najvacsi mozny pocet skokov}
  MAX_SMER = 6;   {pocet smerov, ktorymi sa mozu psici hybat}

type
  TSur = record x,y : integer; end; {suradnice jedneho psika}
  TStav = record p1, p2 : TSur; end; {suradnice dvoch psikov}
  {priestor pre 2 psikov: [x1,y1,x2,y2] hovori, ci tam mozu byt}
```

```
  TMriezka = array [-MAX_V..MAX_V,  -MAX_V..MAX_V,
    -MAX_V..MAX_V,  -MAX_V..MAX_V] of integer;

function dekVrstvu(k : integer) : integer; {cislo vrstvy, kde sa k nachadza}
begin dekVrstvu := trunc(0.5 + sqrt(k/3.0 - 1.0/12.0 - EPS) ); end;

function zakVrstvu(v : integer) : integer; {posledny prvok na danej vrstve}
begin zakVrstvu := 3 * v * v - 3 * v + 1; end;

function dekoduj(k : integer) : TSur; {Dekoduj cislo policka na suradnice}
var v, pv, s, ps : integer;
    sur : TSur;
begin
  v := dekVrstvu(k);
  pv := k - (3 * v * v - 3 * v + 1); {pozicia na vrstve}
  s := pv div v; {strana sestuholnika, kde sa pv nachadza}
  ps := pv mod v; {pozicia na strane sestuholnika}

  case s of
  0 : begin sur.x := +v - 1 - ps; sur.y := +v     ; end;
  1 : begin sur.x :=   -1 - ps; sur.y := +v - 1 - ps; end;
  2 : begin sur.x := -v     ; sur.y :=   -1 - ps; end;
  3 : begin sur.x := -v + 1 + ps; sur.y := -v     ; end;
  4 : begin sur.x :=   +1 + ps; sur.y := -v + 1 + ps; end;
  5 : begin sur.x := +v     ; sur.y :=   +1 + ps; end;
  else writeln('Velka chyba v dekoduj');
  end;
  dekoduj := sur;
end;

var
  n, m, s1, t1, s2, t2 : integer; {vstup}
  v : integer; {max. pouzita vrstva pre dane n}
  A : TMriezka; {prehladavany priestor}
  F : array [0..MAX_STAVOV] of TStav; {fronta}

{zakaze vsetky situacie, kde sa nachadza bodliak na danom mieste}
procedure pridajBodliak( b : TSur );
var x, y : integer;
begin
  for x := -v to v do for y := -v to v do begin
    A[x, y, b.x, b.y] := INF;       {2. psik stoji na bodliaku}
    A[b.x, b.y, x, y] := INF;       {1. psik stoji na bodliaku}
  end;
end;

{vycisti cely prehladavany priestor}
procedure inicializacia;
var x1, y1, x2, y2, i, last : integer;
```

```
begin
   {vycistenie priestoru}
   for x1 := -v to v do for y1 := -v to v do
      for x2 := -v to v do for y2 := -v to v do
         A[x1, y1, x2, y2] := -1;
   {zarazky po okrajoch - pridame umelo bodliaky}
   last := zakVrstvu(v + 1);
   for i := n + 1 to last do pridajBodliak( dekoduj(i) );
   {zakazeme byt obidvom psikom na tom istom mieste}
   for x1 := -v to v do for y1 := -v to v do A[x1, y1, x1, y1] := INF;
end;

{Posunutie suradnice danynm smerom}
function pohyb( a : TSur; s : integer) : TSur;
const smer : array [1..MAX_SMER] of TSur = (
   (x : 0; y : 1), (x : -1; y : 0), (x : -1; y : -1),
   (x : 0; y : -1), (x : 1; y : 0), (x : 1; y : 1) );
begin
   a.x := a.x + smer[s].x; a.y := a.y + smer[s].y; pohyb := a;
end;

function pracuj : integer; {prehladavanie priestoru, kde sa mozu psici nachadzat}
var
   zac : integer;  {pozicia prveho prvku vo fronte}
   kon : integer;  {pozicia za poslednym prvkom vo fronte = volne miesto}
   i, j, vzd : integer;
   p1, p2, q1, q2 : TSur;
   pt1, pt2 : TSur;        {dekodovane pozicie skrys pre psikov}
begin
   zac := 0; kon := 1; {pridame zaciatok do fronty}
   p1 := dekoduj(s1);
   p2 := dekoduj(s2);
   F[zac].p1 := p1; F[zac].p2 := p2;
   A[ p1.x, p1.y, p2.x, p2.y ] := 0;        {zaciname vo vzdial. 0}

   pt1 := dekoduj(t1); pt2 := dekoduj(t2);

   while (zac <> kon) do begin
      {vyberame stav z fronty a najdeme k nemu vzdialenost}
      p1 := F[zac].p1; p2 := F[zac].p2;
      vzd := A[ p1.x, p1.y, p2.x, p2.y ];
      inc(zac);

      {skusame vsetky kombinacie smerov, kam mozu skakat}
      for i := 1 to MAX_SMER do for j := 1 to MAX_SMER do begin
         if (i = j) then continue; {nemozu skakat rovnako}

         {nove pozicie psikov}
         q1 := pohyb( p1, i ); q2 := pohyb( p2, j );
```

```
            {uz videna pozicia resp. zarazka ?}
            if (A[ q1.x, q1.y, q2.x, q2.y ] >= 0 ) then continue;

            {novo objaveny stav -> pridame do fronty}
            A[ q1.x, q1.y, q2.x, q2.y ] := vzd + 1;
            F[kon].p1 := q1; F[kon].p2 := q2;
            inc(kon);

            {nasli sme koncovy stav?}
            if (pt1.x = q1.x) and (pt1.y = q1.y) and (pt2.x = q2.x) and (pt2.y = q2.y)
               then begin pracuj := vzd + 1; exit; end;
         end;
      end;
      pracuj := -1;
end;

var   x, i : integer;
begin
   while (true) do begin
      read( n, m );
      if ( (n = 0) and (m = 0) ) then break;

      v := dekVrstvu(n);
      inicializacia();

      read( s1, t1, s2, t2 );
      for i := 1 to m do begin read( x ); pridajBodliak( dekoduj(x) ); end;
      if (s1 = s2) and (s2 = t2) then x := 0 {specialny pripad} else x := pracuj;
      if (x >= 0) then writeln(x) else writeln('neda sa');
   end;
end.
```

P-III-5

Označme $S = \sum_{i=1}^{N} l_i$ počet dní, ktoré AttoSoft potrebuje na dokončenie všetkých programov. Posledný program teda dokončíme po S dňoch.

Uvažujme nasledujúci algoritmus. Spočítajme pre každý program pokutu, ktorú by sme zaň zaplatili, ak by sme ho dokončili po S dňoch. Program s najmenšou takou pokutou dáme do rozvrhu ako posledný. Ak je to program číslo i, zostáva nám naplánovať všetky zvyšné programy na prvých $S-l_i$ dní, čo urobíme rovnakým spôsobom (t.j. opäť vyberieme ako posledný program s najnižšou pokutou po $S - l_i$ dňoch, atď.)

Správnosť algoritmu ukážeme indukciou vzhľadom na počet programov, ktoré potrebujeme dokončiť. Ak je potrebné dokončiť jeden program, existuje len jediný možný rozvrh, a teda náš algoritmus funguje správne.

Nech teda počet programov, ktoré treba dokončiť je N a nech pre ľubovoľný menší počet programov náš algoritmus funguje správne. Označme G riešenie získané našim algoritmom (v tomto riešení je posledným programom program číslo i). Nech existuje iné, lacnejšie riešenie O, ktoré končí programom číslo j. Ak $i = j$, tak rozdiel medzi G a O musí byť v poradí prvých

$N-1$ programov. Podľa indukčného predpokladu však toto poradie v riešení G je optimálne, preto riešenie O nemôže byť lacnejšie.

V opačnom prípade, vytvorme nový rozvrh O' nasledujúcim spôsobom. Nech rozvrh O dokončuje programy v poradí $o_1, o_2, \ldots, o_N$ a nech $o_k = i$. Podľa rozvrhu O' dokončíme programy v nasledujúcom poradí: $o_1, o_2, \ldots, o_{k-1}, o_{k+1}, \ldots, o_N, i$. Všimnime si, že riešenie O' je nanajvýš také drahé, ako riešenie O. Pokuta za programy $o_{k+1}, \ldots, o_N$ je totiž nižšia ako v riešení O, lebo ich dokončíme skôr (pokuta rastie s počtom dní po termíne). Navyše, pokuta za program i dokončený po S dňoch určite nepresahuje pokutu za program o_N dokončený po S dňoch, keďže program i sme vybrali tak, aby táto pokuta bola najmenšia možná.

Riešenie O' ale nemôže byť lacnejšie ako riešenie G (platí tu ten istý argument, ako v prvom prípade). Preto ani riešenie O nemôže byť lacnejšie ako G. Dokázali sme, že žiadne lacnejšie riešenie ako G neexistuje, riešenie nájdené našim algoritmom je teda optimálne.

V každom kroku algoritmu musíme spočítať príslušnú pokutu pre každý program, ktorý sme ešte nezaradili do rozvrhu. Preto časová zložitosť algoritmu je $O(N^2)$.

```
program attosoft;

const MAXN = 10000;

var a, b, c, d, l : array [1..MAXN] of longint;
    pouzite       : array [1..MAXN] of boolean;
    N, S          : longint;

function cena(prog, den : longint) : longint;
begin
    cena := ((a[prog] * den + b[prog]) * den + c[prog]) * den + d[prog];
end; { cena }

procedure nacitaj;
var i : integer;
begin
    readln(N);
    S := 0;
    for i := 1 to N do begin
        readln(l[i], a[i], b[i], c[i], d[i]);
        S := S + l[i];
    end;
end; { nacitaj }

procedure spocitaj_rozvrh(d : longint);
var minprog : integer;
    min, cc : longint;
    i       : integer;
begin
    { najdi nepouzity program, ktory ma najnizsiu pokutu po d dnoch }
    minprog := -1;
    for i := 1 to N do begin
        if not pouzite[i] then begin
            cc := cena(i, d);
```

```
            if (minprog = −1) or (cc < min) then begin
                min := cc;
                minprog := i;
            end;
        end;
    end;

    if minprog > −1 then begin

        { oznac program ako pouzity }
        pouzite[minprog] := true;

        { zostav zvysok rozvrhu }
        spocitaj_rozvrh(d − l[minprog]);

        { vypis posledny program na konci rozvrhu }
        writeln(minprog);
    end;
end;

var i : integer;

begin
    nacitaj;
    for i := 1 to N do pouzite[i] := false;
    spocitaj_rozvrh(S);
end.
```

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

53. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia III. kola kategórie P

2. súťažný deň

Vydala IUVENTA
pre vnútornú potrebu Ministerstva školstva SR
Sadzba programom LaTeX
Zodpovedný redaktor M. Forišek