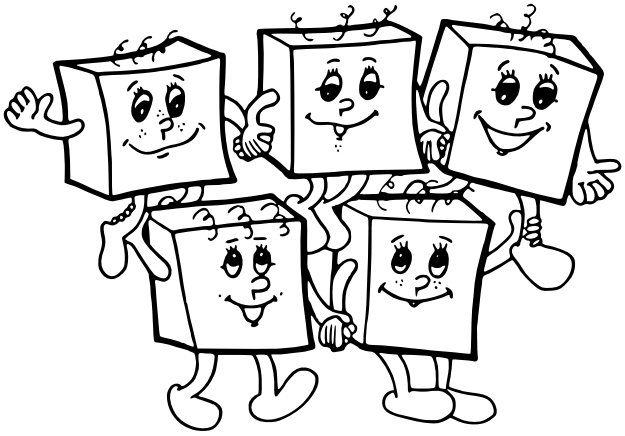
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH

http://oi.sk/



dvadsiaty piaty ročník

školský rok 2009/10

riešenia krajského kola
kategória B

B-II-1 Telemánia

Najjednoduchšie riešenie tejto úlohy je priamočiare. Postupne pre každý z M hovorov spracujeme každého jeho aktéra – teda prezrieme všetkých N zákazníkov, či je to jeden z nich, a ak ho nájdeme, tak pripočítame práve prevolaný počet sekúnd k jeho celkovému času volania. Toto riešenie má časovú zložitosť $O(MN)$.

V tomto riešení najskôr popíšeme niekoľko riešení, ktoré sú lepšie ako toto triviálne, ale nie sú optimálne. Optimálne riešenie nájdete popísané v časti s nadpisom „Vzorové riešenie“.

Myšlienka lepšieho riešenia

Ako naše priamočiare riešenie zlepšiť? Všimnime si, že pre každý hovor vždy znova a znova prechádzame celý zoznam zákazníkov. Keby sme na začiatku tento zoznam upravili do nejakej vhodnejšej podoby, mohli by sme potom vedieť ľahšie zisťovať, či konkrétne telefónne číslo patrí nášmu zákazníkovi (a ak áno, ktorému).

Veľmi dobrý nápad je napríklad všetkých zákazníkov usporiadať podľa telefónneho čísla.

Ako nám toto pomôže? Keď nás teraz bude zaujímať, komu patrí konkrétne telefónne číslo, môžeme použiť binárne vyhľadávanie: Pozrieme sa do prostriedku zoznamu. Ak tam je hľadané číslo, skončili sme, ak nie, vieme, či ho hľadať v prvej alebo v druhej polovici. A tento postup opakujeme, kým číslo buď nenájdeme, alebo nezistíme, že v zozname zákazníkov nie je.

Implementácia lepšieho riešenia

Na triedenie zoznamu zákazníkov môžeme použiť ľubovoľné efektívne triedenie, napr. HeapSort. Toto triedenie je popísané vo vzorových riešeniach domáceho kola. Časová zložitosť tohto algoritmu je $O(N \log N)$, kde N je počet triedených záznamov – v našom prípade teda zákazníkov.

Pozrime sa teraz na jedno binárne vyhľadávanie. Zoznam, v ktorom hľadáme, má N prvkov. Pri binárnom vyhľadávaní v každom kroku zmenšíme počet možností na polovicu. Preto počet krokov, ktoré spravíme, bude rádovo $\log_2 N$.

V našom algoritme spravíme binárne vyhľadávanie $2M$-krát: raz pre každého z účastníkov každého z M hovorov. Táto časť riešenia má teda časovú zložitosť $O(M \log N)$.

Celková časová zložitosť tohto riešenia je teda $O((M + N) \log N)$.

Listing programu:

```
type zakaznik = record
   cislo : string[12];
   meno : string;
   cas : longint;
end;

var N : longint;
    zakaznici : array[1..1000047] of zakaznik;

procedure swap(var x, y : zakaznik);
var z : zakaznik;
begin z:=x; x:=y; y:=z; end;

function neskor(var x, y : zakaznik) : boolean;
begin neskor := x.cislo > y.cislo; end;

procedure insert(i : longint);
begin
   while (i>1) and (neskor(zakaznici[i],zakaznici[i div 2])) do begin
      swap( zakaznici[i div 2], zakaznici[i] );
      i := i div 2;
   end;
end;

procedure extract(pocet : longint);
var i, j : longint;
begin
   swap( zakaznici[1], zakaznici[pocet] );
   dec(pocet);
   i := 1;
   while true do begin
      j := i;
      if (2*i <= pocet) and (neskor(zakaznici[2*i],zakaznici[j])) then j:=2*i;
      if (2*i+1 <= pocet) and (neskor(zakaznici[2*i+1],zakaznici[j])) then j:=2*i+1;
      if j=i then break;
      swap( zakaznici[i], zakaznici[j] );
      i := j;
   end;
end;

procedure nacitaj_zakaznikov;
var c : char;
    i, j : longint;
begin
   readln(N);
   for i:=1 to N do begin
      zakaznici[i].cislo:='';
      for j:=1 to 10 do begin
         read(c);
         zakaznici[i].cislo := zakaznici[i].cislo+c;
      end;
      read(c);
      readln(zakaznici[i].meno);
      zakaznici[i].cas := 0;
   end;
end;

procedure utried_zakaznikov;
var i : longint;
begin
   for i:=2 to N do insert(i);
   for i:=N downto 2 do extract(i);
end;

procedure spracuj_hovor(cislo: string; cas: longint);
var lo, hi, med : longint;
begin
   if cislo < zakaznici[1].cislo then exit;
   { binarnym vyhladavanim najdeme cislo v zozname zakaznikov }
   lo:=1; hi:=N+1;
```

```
   while hi-lo>1 do begin
      med:=(lo+hi) div 2;
      if zakaznici[med].cislo <= cislo then lo:=med else hi:=med;
   end;
   if zakaznici[lo].cislo <> cislo then exit; { nenasiel }
   inc( zakaznici[lo].cas, cas );
end;

procedure spracuj_hovory;
var c : char;
    M, i, j, cas : longint;
    cislo1, cislo2 : string;

begin
   readln(M);
   for i:=1 to M do begin
      cislo1:=''; for j:=1 to 10 do begin read(c); cislo1:=cislo1+c; end;
      read(c);
      cislo2:=''; for j:=1 to 10 do begin read(c); cislo2:=cislo2+c; end;
      read(c);
      readln(cas);
      spracuj_hovor(cislo1,cas);
      spracuj_hovor(cislo2,cas);
   end;
end;

procedure vypis_najlepsieho;
var i, vitaz : longint;
begin
   vitaz := 1;
   for i:=2 to N do if zakaznici[i].cas > zakaznici[vitaz].cas then vitaz:=i;
   writeln(zakaznici[vitaz].cislo+' '+zakaznici[vitaz].meno);
end;

begin
   nacitaj_zakaznikov;
   utried_zakaznikov;
   spracuj_hovory;
   vypis_najlepsieho;
end.
```

Alternatívne, rovnako dobré riešenie

Niektoré programovacie jazyky nám ponúkajú dátovú štruktúru známu pod názvom *asociatívne pole*. Ide o niečo podobné ako klasické pole, ale indexovať do neho môžeme nie len číslami od 0 po nejaké k, ale ľubovoľnými objektmi, ktoré vieme usporiadať.

Ak máme k dispozícii takéto asociatívne polia, môžeme použiť dve: v jednom si ku každému zákazníckemu číslu uložíme meno zákazníka, v druhom jeho počet prevolaných sekúnd.

Konkrétny príklad: v C++ máme v knižnici STL k dispozícii dátovú štruktúru `map`, ktorá predstavuje asociatívne pole.[1] Táto štruktúra je implementovaná pomocou vyvažovaného binárneho stromu, každá operácia s ňou teda trvá čas priamo úmerný logaritmu počtu uložených záznamov.

Riešenie využívajúce `map` má teda tiež časovú zložitosť $O((M+N)\log N)$.

Listing programu:

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
using namespace std;

int main() {
  // nacitame telefonny zoznam
  int N;
  cin >> N;
  vector<string> cisla(N);
  map<string,string> mena;
  for (int n=0; n<N; ++n) { cin >> cisla[n]; cin >> mena[cisla[n]]; }

  // nacitavame hovory a zaznamenavame si ich casy
```

[1] Presnejšie, asociatívne pole, ktoré je navyše usporiadané podľa kľúčov, teda objektov, ktoré používame ako indexy doň.

```
  int M;
  cin >> M;
  map<string,int> casy;
  string cislo1, cislo2;
  int cas;
  for (int m=0; m<M; ++m) {
    cin >> cislo1 >> cislo2 >> cas;
    casy[cislo1] += cas;
    casy[cislo2] += cas;
  }

  // najdeme vitaza
  string vitaz = cisla[0];
  for (int n=1; n<N; ++n) if (casy[cisla[n]] > casy[vitaz]) vitaz = cisla[n];
  cout << vitaz << " " << mena[vitaz] << endl;
}
```

Hešovanie

(Táto časť vzorového riešenia je „nepovinná“ a zaoberá sa komplikovanou metódou, netrpezlivý čitateľ môže preskočiť rovno na časť Vzorové riešenie, kde nájde popis jednoduchšieho a efektívnejšieho riešenia.)

Ešte efektívnejšie riešenie môžeme dosiahnuť použitím hešovania. Trik je v tom, že nepotrebujeme nutne zákazníkov utriediť.

Keby napríklad telefónne čísla boli len 6-ciferné, mohli by sme úlohu ľahko riešiť pomocou obyčajného poľa, do ktorého by sme indexovali priamo telefónnymi číslami.

Základná myšlienka hešovania je v tom, že toto isté chceme robiť aj pre veľké čísla. Ak by sme napríklad našli funkciu f, ktorá pre každé z našich N zákazníckych čísel vráti číslo trebárs od 0 po $3N$ a navyše bude platiť, že pre žiadne dve zákaznícke čísla nedostaneme ten istý výstup, tak sme vyhrali – vždy, keď spracúvame hovor, zoberieme telefónne číslo, funkciou f ho „preložíme“ (zahešujeme) na číslo z malého rozsahu a toto číslo (hešovaciu hodnotu) použijeme ako index do poľa.

Dobrá funkcia f v praxi môže napríklad vyzerať nasledovne: načítame N, nájdeme náhodné prvočíslo p medzi $2N$ a $4N$ a zadefinujeme $f(x) = x \bmod p$.

Napríklad pre vstup zo zadania by sme mohli zvoliť $p = 11$, potom hešovacia hodnota Jožkovho čísla bude $f(0975342583) = 975342583 \bmod 11 = 0$, Ferko bude mať hešovaciu hodnotu 6 a Jano 2.

Ak by všetko fungovalo tak, ako sme to práve popísali, tak máme riešenie v čase $O(N + M)$: Najskôr pre každé z N zákazníckych čísel spočítame jeho hešovaciu hodnotu. Potom pre každého volajúceho aj volaného spočítame hešovaciu hodnotu h jeho čísla. Ak žiadne zákaznícke číslo nemalo hodnotu h, nič nerobíme. Ak ju nejaké malo, pozrieme sa, či sa zhoduje s práve spracúvaným, a ak áno, zvýšime mu prevolaný čas.

Samozrejme, v praxi sa môže stať, že si zvolíme hešovaciu funkciu a následne zistíme, že niektoré dve zákaznícke čísla majú tú istú hešovaciu hodnotu. Toto sa volá *kolízia* a je potrebné to ošetriť – napríklad tak, že pre každú možnú hešovaciu hodnotu si budeme pamätať zoznam všetkých zákazníckych čísel, ktoré ju majú.

Takéto riešenie má, pri dobrej voľbe hešovacej funkcie, *očakávanú* časovú zložitosť $O(N + M)$.

Vzorové riešenie

Teraz ukážeme vzorové riešenie, ktoré bude mať vždy časovú zložitosť $O(N + M)$.

Toto riešenie bude založené na nasledujúcej myšlienke: Keďže telefónne čísla majú konštantný počet cifier, môžeme to využiť a utriediť ich v lineárnom čase. Týmto vylepšením dostaneme algoritmus s časovou zložitosťou $O(N + M \log N)$.

Ale ako sa zbaviť binárneho vyhľadávania? Jednoducho – nahradíme ho ďalším triedením. Prejdeme celý zoznam hovorov a pre každý hovor „číslo1 číslo2 čas“ pridáme do nového poľa dva záznamy: „číslo1 čas“ a „číslo2 čas“. Toto nové pole následne utriedime podľa telefónneho čísla.

Pre príklad zo zadania by toto utriedené pole vyzeralo nasledovne:

```
0955956114 137
0955956114 293
0972125726 137
0972125726 36
0972125726 54
```

```
0972654124 36
0972654124 80
0975342583 293
0975342583 54
0975342583 80
```

No a teraz, keď máme dve utriedené polia (zákazníkov aj hovory), vieme už ľahko zistiť celkové časy hovorov našich zákazníkov. Stačí si všimnúť, že v poli hovorov tvoria hovory každého zákazníka súvislý úsek, a navyše ich čísla sú v tom istom poradí ako v poli so zákazníkmi.

Zostáva ukázať, ako budeme triediť telefónne čísla. Použijeme triedenie známe pod názvom RadixSort. Postupne spravíme 10 prechodov, pričom po k-tom prechode budeme mať čísla utriedené podľa ich *posledných* k cifier.

Pozrime sa bližšie, ako bude vyzerať k-ty prechod. Na začiatku máme čísla utriedené podľa posledných $k-1$ cifier. Pozrieme sa teraz na k-te cifry všetkých čísel, ktoré triedime, a spočítame si, že je medzi nimi c_0 núl, c_1 jednotiek, atď. Teraz vieme, že v poradí utriedenom podľa posledných k cifier budú čísla s nulou na pozíciách 1 až c_0, čísla s jednotkou na pozíciách c_0+1 až c_0+c_1, atď. Tak už len prejdeme zaradom všetky čísla a každé umiestnime na správne miesto do nového poradia.

(Všimnite si, na čo slúžil predpoklad, že čísla už sú utriedené podľa posledných $k-1$ cifier: keď ich ukladáme na nové miesta, tak čísla, ktoré majú na k-tej pozícii rovnakú cifru spracúvame v správnom poradí.)

Každý prechod zjavne prebehne v lineárnom čase, preto je časová zložitosť RadixSortu pre 10-ciferné čísla lineárna od ich počtu.

Celkovo teda toto vzorové riešenie potrebuje $O(N)$ operácií na utriedenie zákazníkov, $O(M)$ na utriedenie hovorov a následne $O(N+M)$ na spracovanie hovorov a vypočítanie výsledku. Celková časová zložitosť je teda $O(N+M)$, čo je zjavne optimálne.

Listing programu:

```
{$H+} { fpc direktiva ktora zapne AnsiStringy, aby zaznamy nezrali zbytocne vela pamate }

type zakaznik = record
   cislo : string[12];
   meno : string;
   cas : longint;
end;

var N, M : longint;
    zakaznici, hovory, pomocne : array[0..2000047] of zakaznik;

procedure radix_sort(var co: array of zakaznik; kolko, cifra: longint);
var i : longint;
    c : array[0..9] of longint;
begin
   for i:=0 to 9 do c[i]:=0;
   for i:=0 to kolko-1 do inc( c[ ord(co[i].cislo[cifra])-48 ] );
   for i:=1 to 9 do c[i]:=c[i-1]+c[i];
   for i:=9 downto 1 do c[i]:=c[i-1]; c[0]:=0;
   for i:=0 to kolko-1 do begin
      pomocne[ c[ ord(co[i].cislo[cifra])-48 ] ] := co[i];
      inc( c[ ord(co[i].cislo[cifra])-48 ] );
   end;
   for i:=0 to kolko-1 do co[i] := pomocne[i];
end;

procedure nacitaj_zakaznikov;
var c : char;
    i, j : longint;
begin
   readln(N);
   for i:=0 to N-1 do begin
      zakaznici[i].cislo:='';
      for j:=1 to 10 do begin
         read(c);
         zakaznici[i].cislo := zakaznici[i].cislo+c;
      end;
      read(c);
      readln(zakaznici[i].meno);
      zakaznici[i].cas := 0;
```

```
   end;
   for i:=10 downto 1 do radix_sort(zakaznici,N,i);
end;

procedure nacitaj_hovory;
var c : char;
    i, j, cas : longint;

begin
   readln(M);
   for i:=0 to M-1 do begin
      hovory[2*i].cislo:='';
      for j:=1 to 10 do begin read(c); hovory[2*i].cislo:=hovory[2*i].cislo+c; end;
      read(c);
      hovory[2*i+1].cislo:='';
      for j:=1 to 10 do begin read(c); hovory[2*i+1].cislo:=hovory[2*i+1].cislo+c; end;
      read(c);
      readln(cas);
      hovory[2*i].cas := cas;
      hovory[2*i+1].cas := cas;
   end;
   for i:=10 downto 1 do radix_sort(hovory,2*M,i);
end;

procedure spracuj;
var z, h : longint;
begin
   z := 0;
   for h:=0 to 2*M-1 do begin
      while (z < N) and (zakaznici[z].cislo < hovory[h].cislo) do inc(z);
      if z = N then break;
      if zakaznici[z].cislo = hovory[h].cislo then
         inc( zakaznici[z].cas, hovory[h].cas );
   end;
end;

procedure vypis_najlepsieho;
var i, vitaz : longint;
begin
   vitaz := 0;
   for i:=1 to N-1 do if zakaznici[i].cas > zakaznici[vitaz].cas then vitaz:=i;
   writeln(zakaznici[vitaz].cislo+' '+zakaznici[vitaz].meno);
end;

begin
   nacitaj_zakaznikov;
   nacitaj_hovory;
   spracuj;
   vypis_najlepsieho;
end.
```

Alternatívne vzorové riešenia

Namiesto RadixSortu od najmenej významnej cifry ho môžeme implementovať aj od najvýznamnejšej pomocou rekurzie. Alebo aj bez rekurzie na dva prechody: rozdelíme si každé číslo na prvých 5 a druhých 5 cifier. Najskôr utriedime čísla podľa ich prvých 5 cifier rovnako, ako v jednej fáze nášho RadixSortu, následne nájdeme úseky s rovnakými prvými 5 ciframi a každý z nich utriedime ešte raz podľa zvyšných 5 cifier. (Rovnako sa dal aj RadixSort zo vzorového riešenia upraviť na 2 prechody namiesto 10.)

Úplne iný spôsob, ako dosiahnuť optimálnu časovú zložitosť, je použiť na zapamätanie zoznamu zákazníkov *písmenkový strom* (trie). Táto dátová štruktúra je popísaná v riešení úlohy A-I-1 z minulého ročníka OI.

B-II-2 Kráľovské cesty

Táto úloha sa dala správne vyriešiť niekoľkými spôsobmi, prejdeme si postupne od menej bodovaných po viac bodované.

Lineárny čas pre jeden telefonát

Toto riešenie bolo asi najjednoduchšie vymyslieť. Spomenieme len stručnú myšlienku. Stačilo si mestá repre-

zentovať ako graf, kde mestá sú vrcholy a cesty medzi mestami sú hrany patričnej dĺžky a potom pri každom telefonáte celý graf prehľadať a nájsť cestu medzi danými mestami.

O niečo lepšie riešenie sa dalo dosiahnuť tak aby sme pri prehľadávaní nepozreli celý graf, ale iba to, čo musíme (teda prejdeme len rádovo toľko vrcholov, koľko ich je na ceste z jedného mesta do druhého). To sa dalo spraviť tak, že z oboch miest pustíme bežcov smerom do hlavného mesta, pričom bežci budú rátať prejdenú vzdialenosť. Ak jeden z bežcov dorazí do druhého mesta, tak skončíme, lebo vzdialenosť miest je práve taká, akoľko prešiel ten bežec. Ak sa bežci stretnú v hlavnom meste, tak je výsledkom súčet prejdených vzdialeností. Toto riešenie má síce v najhoršom prípade časovú zložitosť $O(N)$, ale prakticky je lepšie ako predchádzajúce, lebo často sa stane, že bude potrebovať pozrieť na oveľa menej miest.

Konštantný čas pre jeden telefonát

Konštantný čas sa dal dosiahnuť viacerými spôsobmi, ktoré sa líšili rýchlosťou predspracovania a pamäťovou zložitosťou. Prvý z nich má jednoduchú myšlienku. Vyrobíme si tabuľku $N \times N$, v ktorej budeme mať pre každú dvojicu miest zaznačenú vzdialenosť medzi nimi. Ak už máme tabuľku, tak pri každom telefonáte sa do nej pozrieme a povieme vzdialenosť medzi mestami.

Ako vypočítať takú tabuľku? Najjednoduchší spôsob je pre každú dvojicu miest vypočítať ich vzdialenosť pomocou algoritmu z predchádzajúceho odseku. Takých dvojíc je približne N^2, jedno hľadanie trvá $O(N)$, takže predspracovanie bude trvať $O(N^3)$. Pamäťová zložitosť bude $O(N^2)$, lebo vzdialenosti všetkých dvojíc miest si musíme pamätať.

Tento algoritmus sa dá ľahko vylepšiť tak, že predspracovanie bude trvať $O(N^2)$: stačí pre každý začiatok cesty spustiť prehľadávanie len raz, a keď skončí, pre každé iné mesto si zapamätáme vzdialenosť doň.

Konštantný čas pre telefonát, lineárny pre predspracovanie

V tomto riešení využijeme to, že všetky cesty v kráľovstve začínajú v hlavnom meste a potom sa už vôbec nekrižujú. Predstavme si, že by sme o každom meste vedeli, na ktorej z týchto ciest leží a zároveň by sme vedeli vzdialenosť mesta od hlavného mesta. Teda pre každé mesto a si budeme pamätať hodnotu `a.vzd`, čo je vzdialenosť mesta a od hlavného mesta a hodnotu `a.cesty`, čo je označenie cesty, na ktorej a leží. Ako bude vyzerať jeden telefonát? Dostaneme teraz dve mestá a a b a chceme vedieť ich vzdialenosť. Ak ležia na rôznych cestách (`a.cesta`$\neq$`b.cesta`), tak ak chceme ísť z a do b musíme ísť cez hlavné mesto a preto vzdialenosť miest je súčtom vzdialeností miest a a b od hlavného mesta, čo vieme vypočítať v konštantnom čase.

Ak a aj b ležia na tej istej ceste (`a.cesta`=`b.cesta`), tak ich vzdialenosť bude rozdielom ich vzdialeností od hlavného mesta, čo vieme tiež vypočítať v konštantnom čase. Jedinou otázkou je, ako získať tieto údaje.

Začneme tým, že nájdeme tri mestá, v ktorých jednotlivé cesty končia – toto sú práve tie tri mestá, ktoré nie sú ani raz spomenuté na vstupe (lebo jedine oni nemajú žiadneho suseda, ktorý by od nich bol ďalej).

Keď teraz poznáme mesto, ktorým cesta končí, vieme, že na nej leží aj jeho sused, za ním je sused toho suseda, a tak ďalej, až kým sa nedostaneme ku hlavnému mestu. Toto spravíme pre každú z ciest samostatne.

A keď už pre cestu poznáme všetky mestá, ktoré na nej ležia, môžeme po nej prejsť ešte raz – tentokrát v opačnom smere – a pre každé z nich zistiť, ako je ďaleko od hlavného mesta.

Listing programu:

```
type mesto = record
   sused, vzdialenost : longint; { udaje zo vstupu }
   cesta, do_hlavneho : longint; { cislo cesty a vzdialenost do hl. m. }
   koniec : boolean;             { konci v tomto meste cesta? }
end;

var N : longint;
    mesta : array of mesto;
    dlzky : array[1..3] of longint;          { dlzky jednotlivych ciest }
    cesty : array[1..3] of array of longint; { cisla miest na nich }
    f : text;
    i,x,y : longint;

begin
   { nacitame mesta }
   assign(f,'mapa.txt'); reset(f);
```

```
   readln(f,N);
   setlength(mesta,N);
   mesta[0].do_hlavneho := 0;
   for i:=1 to N-1 do mesta[i].koniec := true;
   for i:=1 to N-1 do begin
      readln(f,x,y);
      mesta[i].sused := x-1;
      mesta[i].vzdialenost := y;
      mesta[x-1].koniec := false;
   end;

   { zistime kde koncia cesty a zostrojime ich }
   for i:=1 to 3 do setlength(cesty[i],N);
   x := 0;
   for i:=1 to N-1 do
      if mesta[i].koniec then begin
         inc(x);
         cesty[x][0] := i;
         y := 1;
         while true do begin
            cesty[x][y] := mesta[ cesty[x][y-1] ].sused;
            if cesty[x][y] = 0 then break;
            inc(y);
         end;
         dlzky[x] := y;
      end;

   { prejdeme po kazdej ceste od hlavneho mesta a zistime do_hlavneho }
   for x:=1 to 3 do begin
      for y:=dlzky[x]-1 downto 0 do begin
         mesta[ cesty[x][y] ].cesta := x;
         mesta[ cesty[x][y] ].do_hlavneho :=
            mesta[ cesty[x][y+1] ].do_hlavneho + mesta[ cesty[x][y] ].vzdialenost;
      end;
   end;

   { odpovedame na otazky }
   while not eof do begin
      readln(x,y); dec(x); dec(y);
      if mesta[x].cesta = mesta[y].cesta then
         writeln( abs( mesta[x].do_hlavneho - mesta[y].do_hlavneho ) )
      else
         writeln( mesta[x].do_hlavneho + mesta[y].do_hlavneho );
   end;
end.
```

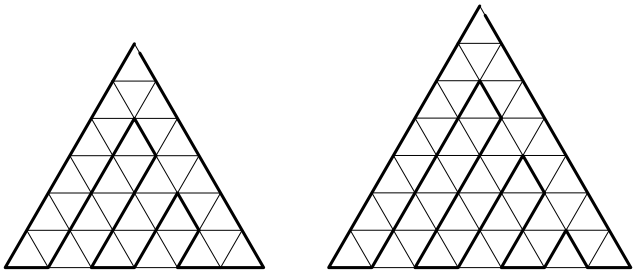
B-II-3 Triplex

Každú podúlohu budeme riešiť samostatne, obe riešenia ale budú mať jedno spoločné: Kľúčom k úspechu bude zvoliť si z mnohých možných riešení také, ktoré sa nám bude čo najjednoduchšie programovať.

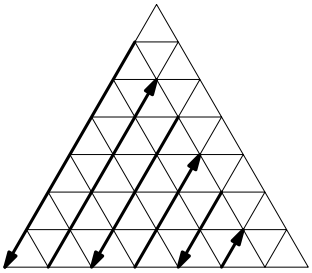
Podúloha a: prejsť všetky vrcholy

V teórii grafov sa trasa, ktorú má naša figúrka prejsť (teda trasa prechádzajúca práve raz cez každý vrchol) volá *Hamiltonovská kružnica*. Vo všeobecnosti je jej hľadanie veľmi ťažký problém a nie je preň známy žiaden efektívny algoritmus. My ale našťastie máme veľmi špecifický hrací plán, pre ktorý to vždy ide.

Riešenie, ktoré sme si vybrali, si najskôr ukážeme pre $N = 6$ a $N = 7$:



Toto riešenie vieme veľmi ľahko zovšeobecniť pre ľubovoľné N. Všimnime si na nasledujúcom obrázku, ako presnejšie funguje „cik-cak“ časť našej cesty:



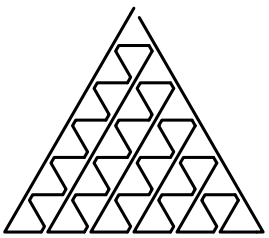
Znázornené šípky sú postupne: $N-1$ krokov dole vľavo, $N-2$ krokov hore vpravo, $N-3$ krokov dole vľavo, a tak ďalej. Toto ľahko prepíšeme do cyklu.

Jediné, na čo si ešte potrebujeme dať pozor, je úplný koniec. Všimnite si, že v okolí pravého dolného rohu sa riešenia pre $N = 6$ a $N = 7$ mierne líšia. Rovnako ako $N = 6$ bude vyzerať riešenie pre každé párne N a rovnako ako $N = 7$ bude vyzerať riešenie pre každé nepárne N.

Listing programu:

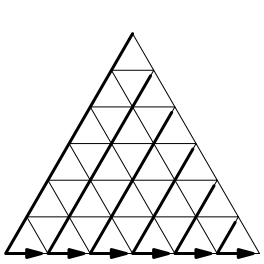
```
var N, i, j : longint;
begin
   readln(N);
   write(4);
   for i:=1 to N-1 do begin
      if i mod 2 = 1 then begin
         for j:=1 to N-i do write(4);
         write(0);
      end else begin
         for j:=1 to N-i do write(1);
         write(5);
      end;
   end;
   write(0);
   for i:=1 to N do write(2);
   writeln;
end.
```

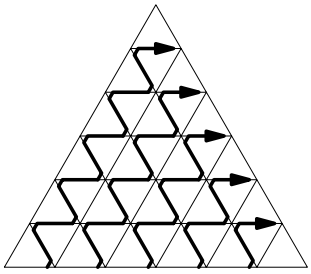
Podúloha b: prejsť všetky hrany



V teórii grafov sa tento druh trasy volá *Eulerovský ťah*. Na rozdiel od Hamiltonovskej kružnice, na hľadanie Eulerovského ťahu existujú všeobecné efektívne algoritmy. Jedna možná myšlienka je, že začneme s nejakou trasou, ktorá ide každou hranou najviac raz (napríklad prázdnou), a kým neobsahuje všetky hrany, tak dokola opakujeme: nájdeme nejaký vrchol, v ktorom ešte máme nepoužitú hranu. Tam našu trasu „rozpojíme“ a vložíme do nej nový kus, ktorý začína aj končí v uvedenom vrchole.

Nebudeme tu ale zachádzať do detailov, pretože pre našu úlohu existuje aj jednoduchšie riešenie – opäť si stačí zvoliť vhodnú, pravidelnú trasu. Napríklad tú, ktorá je znázornená na obrázku vpravo. V tejto trase sa striedajú presuny priamo doľava dole a presuny „cik-cak“ naspäť doprava hore. Lepšie to uvidíme, keď si trasu prekreslíme do dvoch obrázkov:





Listing programu:

```
var N, i, j : longint;
begin
   readln(N);
   for i:=N downto 1 do begin
      for j:=1 to i do write(4);
      write(0);
      for j:=1 to i-1 do write(20);
   end;
   for i:=1 to N do write(2);
   writeln;
end.
```

B-II-4 Gaštanový snehuliak

Podobne ako v domácom kole ukážeme, že pre Tomáška existuje stratégia, pri ktorej bude počet otázok, ktoré mu stačia, lineárny od počtu gaštanov N.

Základné úvahy

Základný „stavebný kameň“ nášho riešenia bude taktiež rovnaký: Tomáško si vyberie nejaký gaštan a opýta sa $N-1$ otázok, aby zistil, s koľkými a ktorými inými gaštanmi je ten jeho gaštan spojený.

Gaštan nazveme *obyčajný*, ak patrí do práve jednej z troch gulí nášho snehuliaka a nie je spojený s nosom. Sú práve štyri gaštany, ktoré nie sú obyčajné: jeden je nos, druhý je jeho jediný sused, tretí je spoločný gaštan hornej a strednej a štvrtý je spoločný gaštan strednej a dolnej gule. Tieto štyri gaštany budeme volať *špeciálne*.

Pre daný gaštan g nazveme *okolím* g množinu obsahujúcu gaštan g a všetky gaštany, ktoré sú s ním spojené zápalkou. Zistiť ako vyzerá okolie gaštanu g vieme položením $N-1$ otázok.

Oplatí sa, skôr než začneme hrať hru, spraviť nasledujúce pozorovania:

- Žiadne dva špeciálne gaštany nemajú rovnaké okolie.
- Žiaden obyčajný gaštan nemá rovnaké okolie ako žiaden špeciálny gaštan.
- Dva obyčajné gaštany majú rovnaké okolie práve vtedy, ak patria do tej istej gule.

Riešenie

Budeme postupovať nasledovne. Najskôr nájdeme jednu guľu:

- Vyberieme si ľubovoľných 8 gaštanov. Ku každému z nich nájdeme jeho okolie.
- Vyberieme spomedzi našich gaštanov dva, ktoré majú rovnaké okolie A.
  (Keďže len 4 gaštany sú špeciálne, medzi našimi 8 gaštanmi sú aspoň 4 obyčajné. A keďže snehuliak má len tri gule, z niektorej gule máme aspoň dva obyčajné gaštany. A tieto dva gaštany majú určite rovnaké okolie.)
- Okolie A je presne jedna z gulí nášho snehuliaka.

Na $8(N-1)$ otázok sme teda našli prvú guľu. Nevieme ešte, ktorá to je, ale vieme, že určite obsahuje aspoň jeden špeciálny gaštan. Na nájdenie druhej gule nám teda bude stačiť aj menej otázok:

- Vyberieme si ľubovoľných 6 gaštanov, ktoré nepatria do A. Ku každému z nich nájdeme jeho okolie.
- Vyberieme spomedzi našich gaštanov dva, ktoré majú rovnaké okolie B.
- Okolie B je druhá z gulí nášho snehuliaka.

A pre veľký úspech postup zopakujeme aj do tretice.

- Vyberieme si ľubovoľné 4 gaštany, ktoré nepatria do A ani do B. Ku každému z nich nájdeme jeho okolie.
  (V tomto kroku môže nastať jeden špeciálny prípad: ak sú A a B horná a dolná guľa, a ak je stredná guľa tvorená len 4 gaštanmi, ostali nám už len tri neidentifikované gaštany. V takomto prípade vyberieme len tieto tri.)

- Vyberieme spomedzi našich gaštanov dva, ktoré majú rovnaké okolie C.
- Okolie C je tretia z gulí nášho snehuliaka.

Podľa toho, ktoré dvojice okolí A, B a C majú neprázdny prienik, vieme povedať, ktorá guľa je stredná – tá, ktorá má spoločný gaštan s každou zo zvyšných dvoch.

A ešte nám ostal jeden gaštan, ktorý nepatrí do A, B, ani C. Tento gaštan je nos. A tá množina spomedzi A, B a C, ktorá obsahuje jeho suseda, je horná guľa.

Pre ľubovoľne veľké N si náš algoritmus vystačí s menej ako $18N$ otázkami, je teda lineárny od počtu gaštanov.

Alternatívne riešenie

Existujú aj riešenia, ktoré si vystačia s o niečo menej otázkami ako to naše. Výhodou nášho riešenia ale je, že sa ľahko vysvetľuje a neobsahuje (takmer) žiadne špeciálne prípady.

Uvedieme ešte jeden iný spôsob, ako sa dalo túto hru vyhrať s lineárnym počtom otázok. Základná myšlienka tohto spôsobu je, že najskôr nájdeme nos, a od nosa potom postupne zostrojíme snehuliaka.

Najskôr popíšeme druhú časť riešenia. Akonáhle niekedy vieme, ktorý gaštan je nos, môžeme postupovať nasledovne: Jeho jediný sused patrí do hornej gule. Hornú guľu tvorí jeho okolie okrem nosa. Teraz vyberieme dva zo zvyšných gaštanov a ku každému nájdeme jeho okolie. Aspoň jeden z nich je obyčajný, a teda jeho okolie bude niektorá iná guľa. Podľa toho, či má spoločný gaštan s hornou, vieme, či je to stredná alebo dolná guľa. V oboch prípadoch si na záver vyberieme jeden ľubovoľný zo zvyšných gaštanov a nájdeme jeho okolie, aby sme presne vedeli aj tretiu guľu.

Jediné, čo ešte treba vymyslieť, je ako nájsť snehuliakov nos. Zjavne akonáhle nájdeme okolie nejakého gaštanu a zistíme, že má len jedného suseda, vieme, že je to nos. Rozoberieme teda len situácie, kedy žiaden z gaštanov, ktoré si vyberieme, nebude nos.

Nos budeme hľadať nasledovne: Vyberieme si prvý gaštan, nájdeme jeho okolie. Vyberieme si druhý gaštan, ktorý neleží v okolí prvého, a nájdeme jeho okolie.

Teraz mohli nastať tri prípady:

1. Obe okolia dokopy pokrývajú všetky gaštany okrem jedného. Vtedy ten jeden gaštan musí byť nos.
2. Obe okolia dokopy pokrývajú úplne všetky gaštany. Toto mohlo nastať jedine tak, že sme trafili dva špeciálne gaštany – suseda nosa a gaštan spoločný pre strednú a spodnú guľu. Vyberieme teda namiesto nich dva iné gaštany a už máme istotu, že tento prípad nenastal.
3. Ostalo viac gaštanov, ktoré do ani jedného z nájdených okolí nepatria. Tento prípad rozoberieme nižšie.

Teraz vyberieme jeden zo zvyšných vrcholov a nájdeme jeho okolie. A už sú len dva možné prípady. V tom lepšom zostal jeden gaštan, ktorý nesusedí so žiadnym z našich troch, a to musí byť nos. V tom horšom prípade naše tri okolia pokrývajú úplne všetky gaštany, lebo jeden z nich je sused nosa. Ak teda nastal tento prípad, tak postupne každý z troch našich gaštanov skúsime nahradiť iným, ktorý neleží v okolí zvyšných dvoch.

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY PIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX