Dvadsiaty ôsmy ročník Olympiády v informatike

SLOVENSKÁ INFORMATICKÁ SPOLOČNOSŤ

Odborným garantom súťaže Olympiáda v informatike je Slovenská informatická spoločnosť. Viac sa o nej dozviete na stránke `http://www.informatika.sk/`

Na obálke ročenky je tzv. oblak slov (word cloud) tvorený najčastejšie sa vyskytujúcimi slovami z textu minulej ročenky. Veľkosť písma približne zodpovedá logaritmu počtu výskytov daného slova.

Oblaky slov síce asi nemajú žiadne vedecké využitie, ide však o veľmi účinnú formu vizualizácie dát – človek dokáže v priebehu niekoľkých sekúnd pochopiť podstatnú časť ich zloženia.

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty ôsmy ročník Olympiády v informatike
ISBN 978-80-8072-145-9

Obsah

O priebehu 28. ročníka Olympiády v informatike 3

Zadania domáceho kola kategórie A 5
Zadania domáceho kola kategórie B 15
Zadania krajského kola kategórie A 21
Zadania krajského kola kategórie B 28
Zadania celoštátneho kola kategórie A 35

Riešenia domáceho kola kategórie A 47
Riešenia domáceho kola kategórie B 62
Riešenia krajského kola kategórie A 77
Riešenia krajského kola kategórie B 92
Riešenia celoštátneho kola kategórie A 103

Výsledky krajských kôl kategórie B 126
Výsledky celoštátneho kola kategórie A 128
Výsledky výberového sústredenia 129

Medzinárodné prípravné sústredenie v Davose 130
Česko-poľsko-slovenské prípravné sústredenie 132
Stredoeurópska olympiáda v informatike 133
Medzinárodná olympiáda v informatike 134

O priebehu 28. ročníka Olympiády v informatike

V školskom roku 2012/13 na Slovensku prebehol už dvadsiaty ôsmy ročník Olympiády v informatike (OI).

Do súťaže sa zapojilo 62 žiakov v kategórii A (starší) a 36 v kategórii B (mladší). Najlepších 29 riešiteľov kategórie A sa zúčastnilo celoštátneho kola, ktoré sa tohto roku konalo v Košiciach.

Vynikajúco sa darilo našim najúspešnejším riešiteľom na medzinárodných súťažiach: najskôr priniesli z Medzinárodnej olympiády v informatike 2 zlaté, striebornú a bronzovú medailu (čo je najlepší výsledok Slovenska od roku 2005) a následne zlato za absolútne víťazstvo a bronz zo Stredoeurópskej olympiády v informatike.

Na celoslovenskej úrovni sa počas celého ročníka o plynulý chod OI starala Slovenská komisia OI v nasledujúcom zložení:

- doc. RNDr. Gabriela Andrejková, CSc.,
  predsedkyňa, ÚI PF UPJŠ, Košice
- RNDr. Michal Forišek, PhD., podpredseda, KI FMFI UK, Bratislava
- RNDr. Andrej Blaho, PhD., FMFI UK, Bratislava
- Mgr. Vladimír Boža, FMFI UK, Bratislava
- Michal Anderle, FMFI UK, Bratislava
- PaedDr. Ivan Brodenec, KI FPV UMB, krajský predseda pre BB
- RNDr. Eva Hanulová, Gym. J. Hronca, krajská predsedkyňa pre BA
- RNDr. Rastislav Krivoš-Belluš, PhD.,
  Ústav informatiky PF UPJŠ, krajský predseda pre KE
- prof. Ing. Veronika Stoffová, CSc.,
  KI PF UJS, krajská predsedkyňa pre NR
- Mgr. Mária Majherová, PhD.,
  Katedra matematiky FHPV PU, krajská predsedkyňa pre PO
- Ing. Andrea Julény, KI FMech TnUAD, krajská predsedkyňa pre TN
- doc. RNDr. Mária Lucká, CSc.,
  KMI PdF TU, krajská predsedkyňa pre TT

- RNDr. Peter Varša, PhD., KI FRI ŽU, krajský predseda pre ZA

Zástupcom IUVENTY zabezpečujúcim organizačnú stránku súťaže bola Mgr. Mária Gajarová.

Michal Forišek, podpredseda SK OI

Zadania domáceho kola kategórie A

A-I-1 Špióni

Kontrarozviedke v Absurdistane sa podaril husársky kúsok: odhalili celú sieť nepriateľských špiónov v krajine.

Niektoré dvojice špiónov spolu komunikujú. Z bezpečnostných dôvodov je táto sieť pomerne riedka – komunikujúcich dvojíc je málo. Presnejšie, pre ľubovoľné prirodzené číslo k platí: ak ľubovoľným spôsobom vyberieme k špiónov, bude medzi nimi najviac $3k$ dvojíc špiónov, ktorí spolu komunikujú.

Súťažná úloha:

Nájdite v danej sieti špiónov najväčšiu *kliku*: skupinu špiónov, v ktorej spolu komunikuje každá dvojica.

(Pomôcka: Z riedkosti komunikačnej siete vyplýva, že určite neexistuje klika tvorená viac ako 7 špiónmi.)

Formát vstupu:

V prvom riadku vstupu sú dve celé čísla n a m. Číslo n udáva počet špiónov, číslo m je počet dvojíc, ktoré spolu komunikujú. Špióni majú pridelené čísla od 1 po n.

Každý z nasledujúcich m riadkov obsahuje dve rôzne čísla od 1 po n, popisujúce jednu dvojicu špiónov, ktorá spolu komunikuje. (Žiadne dva riadky nepopisujú tú istú dvojicu špiónov.)

Môžete predpokladať, že vstup popisuje riedku komunikačnú sieť spĺňajúcu podmienku zo zadania. Špeciálne teda platí $0 \leq m \leq 3n$.

Formát výstupu:

Na výstup vypíšte jediný riadok a v ňom niekoľko navzájom rôznych čísel z rozsahu od 1 po n: čísla špiónov tvoriacich najväčšiu kliku.

Čísla môžete vypísať v ľubovoľnom poradí. Ak existuje viacero najväčších klík, stačí nájsť a vypísať ľubovoľnú jednu z nich.

Hodnotenie:

Vo vstupoch, za ktoré môžete získať 2 body, bude platiť $1 \leq n \leq 40$.

Ďalších 5 bodov môžete získať za vstupy, v ktorých platí: $1 \leq n \leq 1000$ a zároveň celkový počet klík špiónov (všetkých, nie len maximálnych) neprevýši 64 000.

Vo všetkých vstupoch platí $1 \leq n \leq 100\,000$.

Príklad:

Vstup

```
5 6
1 2
1 3
1 4
1 5
3 2
4 2
```

Výstup

```
1 2 3
```

Iným správnym výstupom by bolo „4 1 2“.
Výstup „1 2 3 4“ nie je správny, lebo špióni 3 a 4 spolu nekomunikujú, táto skupina teda nie je klikou.
Výstup „1 5“ nie je správny, lebo to síce je klika, ale nie je najväčšia možná.

A-I-2 Rozvod elektriny

Za siedmimi horami a za siedmimi dolinami vyšla z domčeka ježibaba a povedala: „Prečo práve ja musím bývať tak ďaleko? Ani elektrinu sem ešte nedotiahli!“

Po tom, ako sa ježibaba sťažovala na krajskom úrade, rozhodli sa radní, že je načase zaviesť do celého kraja elektrinu (skôr, ako sa niekto z nich čisto náhodou premení na ropuchu). V prvej fáze výstavby rýchlo postavili v kraji niekoľko elektrární. Potom prišla druhá fáza: V kraji bolo n lokalít (miest a elektrární), ktoré potrebovali pospájať elektrickým vedením do rozvodnej siete.

Toto spájanie spravili systematicky, aby sa na nič nezabudlo. Začali tým, že ku krajskému mestu priamym vedením pripojili jednu elektráreň. Takto dostali základ rozvodnej siete, ktorú následne rozširovali. Vždy si vybrali jednu lokalitu, ktorá ešte nebola pripojená, a postavili vedenie medzi ňou a jednou z lokalít, ktoré už pripojené boli. Takto vznikla rozvodná sieť so stromovou topológiou. V niektorých uzloch tejto siete boli mestá, v ostatných zas elektrárne.

Súťažná úloha:

Ukázalo sa, že sieť treba zase rozpojiť na niekoľko častí, pričom v každej bude niekoľko miest a práve jedna elektráreň. Totiž ako tak narýchlo stavali elektrárne, stalo sa, že každá z nich produkuje striedavý prúd s inou frekvenciou. Preto si každé mesto musí vybrať práve jednu elektráreň, z ktorej bude

brať elektrinu. Vašou úlohou bude nájsť toto priradenie miest k elektrárňam. Presnejšie, treba dodržať nasledovné podmienky:

- Jedným vedením sa nedá prenášať elektrickú energiu z dvoch rôznych elektrární.
- Nevieme cez mesto pripojené k jednej elektrárni preniesť elektrinu z inej elektrárne, ani cez jednu elektráreň preniesť elektrinu z inej elektrárne.
- Každá elektráreň má svoj výkon – množstvo energie, ktoré vie za deň vyrobiť. A naopak, každé mesto má svoju spotrebu – množstvo energie, ktorú za deň spotrebuje.
- Každé vedenie má svoju kapacitu – maximálne množstvo energie, ktorú ním za deň vieme preniesť.

Formát vstupu:

V prvom riadku sú celé čísla n a d_1. Číslo n je počet lokalít (miest a elektrární dokopy). Lokality sú očíslované od 1 po n v poradí, v akom boli pripojené k rozvodnej sieti. Číslo 1 má teda krajské mesto a číslo 2 nejaká elektráreň. Číslo d_1 je spotreba krajského mesta.

Nasledujúce riadky popisujú ostatné lokality. Presnejšie, i-ty riadok vstupu (pre $2 \leq i \leq n$) popisuje lokalitu i. Popis má tvar „$z_i\ m_i\ c_i\ d_i$“. Znak z_i je buď „`M`“ alebo „`E`“, podľa toho, či ide o mesto alebo o elektráreň. Nasledujú tri celé čísla: m_i ($1 \leq m_i < i$) je číslo lokality, ku ktorej sme lokalitu i pripojili, keď sme budovali sieť. Kapacita vedenia medzi lokalitami i a m_i je c_i. No a hodnota d_i má dva možné významy: ak ide o mesto, d_i je jeho spotreba, ak ide o elektráreň, d_i je jej výkon.

Môžete predpokladať, že platí $2 \leq n$ a že pre všetky i platí $0 \leq c_i, d_i \leq 10^9$.

Hodnotenie:

Vo všetkých testovacích vstupoch bude platiť $n \leq 1\,000\,000$.
V testovacích vstupoch za 5 bodov bude platiť $n \leq 1\,000$.

Formát výstupu:

Ak nie je možné priradiť mestám elektrárne tak, aby boli splnené všetky požiadavky, vypíšte jeden riadok a v ňom text „`nemozne`“. V opačnom prípade vypíšte jeden riadok a v ňom n čísel: $a_1, \ldots, a_n$. Ak je i-ta lokalita elektráreň, má byť $a_i = i$. Ak je to mesto, a_i má byť číslo elektrárne, z ktorej bude mesto i brať elektrinu.

Ak existuje viacero možných riešení, vypíšte ľubovoľné z nich.

(Nezabudnite: Pre každú elektráreň musí jej výkon byť väčší alebo rovný ako súčet spotreby miest, ktoré sú ku nej pripojené. Pre každé mesto musí platiť, že všetky mestá, ktoré ležia na ceste medzi ním a elektrárňou, ku ktorej je pripojené, sú pripojené k tej istej elektrárni. Pre každé vedenie, ktoré spája dve lokality pripojené k tej istej elektrárni, musí platiť: súčet spotrieb miest, ktoré by boli prerušením tohto vedenia oddelené od elektrárne, nesmie presiahnuť kapacitu dotyčného vedenia.)

Príklad:

Vstup

```
5 5
E 1 5 10
E 1 5 20
M 3 15 10
M 4 5 5
```

Výstup

```
2 2 3 3 3
```

Vstup

```
5 5
E 1 5 10
E 1 5 20
M 3 12 10
M 4 5 5
```

Výstup

```
nemozne
```

A-I-3 Húsenice

Vo Vihorlatskom pralese sa nedávno rozmnožili nebezpečné húsenice. Ohryzávajú stromy a hrozí, že onedlho bude z pralesa už len obyčajný les. Ako to však často býva, príroda si poradí aj sama. V pralese totiž žije škorec Ignác, ktorý práve vyráža na lov. Keď sa vydá na lov, letí po polpriamke zo svojho hniezda až na kraj pralesa a zožerie všetky húsenice, ponad ktoré preletí.

Súťažná úloha:

Pre jednoduchosť si prales predstavíme ako vodorovnú rovinu a jednotlivé húsenice ako úsečky v tejto rovine. Ignác má hniezdo v bode $(0, 0)$. Nájdite polpriamku, po ktorej má Ignác letieť, ak chce húseníc uloviť najviac.

Formát vstupu:

V prvom riadku je počet úsečiek n. Nasleduje n riadkov, každý z nich popisuje jednu úsečku. V i-tom z týchto riadkov sú štyri celé čísla $x_{i,1}$, $y_{i,1}$, $x_{i,2}$ a $y_{i,2}$. Tieto určujú úsečku s koncovými bodmi $(x_{i,1}, y_{i,1})$ a $(x_{i,2}, y_{i,2})$.

Úsečky sa môžu navzájom pretínať aj prekrývať. Taktiež je možné, že niektorými bodmi roviny prechádzajú viac ako dve úsečky. Niektoré súradnice koncových bodov úsečiek môžu byť aj záporné.

Môžete ale predpokladať, že pre každú úsečku platí podmienka $x_{i,1}y_{i,2} \neq x_{i,2}y_{i,1}$. (Z tejto podmienky vyplýva, že každá úsečka má kladnú dĺžku a tiež že žiadna úsečka neprechádza hniezdom škorca Ignáca).

Formát výstupu:

Vypíšte jeden riadok a v ňom dve celé čísla x a y. Tieto čísla nesmú byť obe nulové. Tieto čísla sú súradnice bodu, ktorým prechádza polpriamka, po ktorej má škorec Ignác letieť. Počet húseníc, ktoré pretne táto polpriamka, musí byť maximálny možný. Ak je viacero optimálnych riešení, vypíšte ľubovoľné z nich.

Hodnotenie:

Plných 10 bodov dostanete za riešenie, ktoré dokáže efektívne vyriešiť vstup obsahujúci stotisíce húseníc. Až 7 bodov môžete získať za riešenie, ktoré dokáže efektívne vyriešiť vstup s tisícmi húseníc.

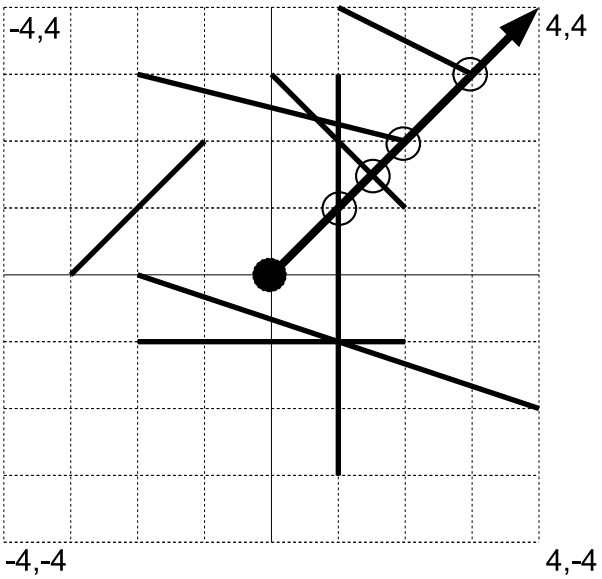
Príklad:

Vstup

```
7
1 4 3 3
0 3 2 1
1 3 1 -3
2 -1 -2 -1
-2 0 4 -2
-3 0 -1 2
-2 3 2 2
```

výstup

```
4 4
```



Existuje viacero správnych riešení, okrem bodu $(4, 4)$ *vyhovujú napr. aj body* $(10, 10)$ *a* $(3, 4)$. *Vo všetkých týchto prípadoch Ignác uloví 4 húsenice.*

A-I-4 Log-space výpočty

V tomto ročníku olympiády sa budeme v každom súťažnom kole stretávať s programami, ktoré môžu používať len veľmi malé množstvo pamäte. V študijnom texte uvedenom za zadaním tejto úlohy sú popísané obmedzenia, ktoré musíte pri riešení tejto úlohy dodržať.

Súťažná úloha:

Pri nasledujúcich úlohách nám vôbec nezáleží na časovej zložitosti vašich programov. Nemusíte ju ani odhadovať. Každý korektný program, spĺňajúci požiadavky uvedené v študijnom texte, dostane plný počet bodov.

a) (4 body) Na vstupe je číslo n a pole $A[1..n]$ obsahujúce postupnosť čísel. Napíšte log-space program, ktorý do výstupného poľa $B[1..n]$ vyplní túto postupnosť usporiadanú od najmenšieho čísla po najväčšie. (Pozor, niektoré čísla sa v poli A môžu opakovať!)

   Teda napr. pre vstupné pole $A = (3, 1, 4, 1, 5)$ máte vyrobiť výstupné pole $B = (1, 1, 3, 4, 5)$.

b) (4 body) Na vstupe je číslo n a dve polia $A[1..n]$ a $B[1..n]$ obsahujúce postupnosti čísel. Napíšte log-space program, ktorý zistí, či sa tieto postupnosti líšia len poradím prvkov.

   Teda napr. pre polia $A = (3, 1, 4, 1, 5)$ a $B = (1, 4, 5, 1, 3)$ je odpoveď „áno“, pre $A = (1, 2, 2)$ a $B = (2, 1, 1)$ je odpoveď „nie“.

c) (2 body) Majme dva log-space programy $\mathcal{F}$ a $\mathcal{G}$. Program $\mathcal{F}$ dostane vstup v poli $A[1..n]$ a vyrobí z neho výstupné pole $B[1..n]$. Program $\mathcal{G}$ dostane na vstupe pole B, ktoré vyrobil program $\mathcal{F}$, a svoj výstup zapíše do výstupného poľa $C[1..n]$.

   Dokážte, že existuje log-space program $\mathcal{H}$, ktorý na vstupe dostane pole A a na výstupe vyrobí zodpovedajúce pole C.

   (V čom je problém? Keby nám nezáležalo na pamäťovej zložitosti, mohli by sme najskôr ako podprogram spustiť $\mathcal{F}$, vypočítať si B, a potom spustiť $\mathcal{G}$ a vypočítať tak C. Problém je v tom, že na takéto riešenie nemáme dosť pamäte – nezmestí sa nám do nej pole B.)

Študijný text

Ak poznáme viac algoritmov, ktoré riešia tú istú úlohu, väčšinou za lepší považujeme ten, ktorý má menšiu časovú zložitosť. Pamäťovú zložitosť zväčša používame len ako dodatočné kritérium (s výnimkou situácií, kedy sú pamäťové nároky algoritmu absurdne vysoké). V úlohách, ku ktorým patrí tento študijný text, bude situácia presne opačná: zaujímať nás bude takmer výlučne pamäťová zložitosť programu. Tá bude musieť byť veľmi malá.

Presnejšie, budeme písať *log-space programy*. Pôjde o obyčajné programy vo vašom obľúbenom bežnom programovacom jazyku, len budú navyše musieť spĺňať nasledujúce obmedzenia:

- Každá použitá premenná musí byť jednoduchého *celočíselného* typu (napr. `int` v C++ či `longint` v Pascale).
- U celočíselných premenných nám nebude záležať na presnom rozsahu, nemusí vás teda napr. trápiť, či sú 32-bitové alebo 64-bitové. Namiesto toho si povolený rozsah hodnôt, ktoré sa do premenných zmestia, definujeme nasledovne: Nech n je veľkosť vstupu (teda napr. počet čísel na vstupe). Potom do premenných môžeme ukladať len hodnoty ktoré sú *polynomiálne veľké* v závislosti od n.

  Môžeme mať teda napr. premennú, ktorá bude nadobúdať hodnoty $-n \ldots n$, hodnoty $-3n^5 \ldots 3n^5$, čí len hodnoty $-4 \ldots 7$. Nesmieme však už mať premennú nadobúdajúcu napr. hodnoty $0 \ldots 2^n$.
- Pre pohodlie budeme aj typy `char` a `boolean` (resp. `bool` v C++) považovať za celočíselné, a teda povolené.
- Žiadne iné typy premenných (teda napr. ani polia alebo ukazovatele) nie sú povolené.
- Výnimku z predchádzajúcich pravidiel tvoria vstup a výstup. Vstup programu bude dostupný v nejakých špeciálnych premenných (zväčša to budú polia), ktoré môže váš program *len čítať*. A podobne výstup bude váš program ukladať do iných špeciálnych premenných, do ktorých smie *len zapisovať*.

  (Špeciálne upozorňujeme, že nesmiete výstupnú premennú zväčšiť o danú hodnotu. Teda na ňu nesmiete napr. v Pascale použiť príkaz `inc`, v C++ operátor `++` či `+=`. Všetky tieto zmeny totiž vyžadujú nie len zápis novej hodnoty, ale najskôr aj prečítanie starej – lenže výstupnú premennú čítať nesmiete.)

Čísla na vstupe vždy budú mať veľkosť polynomiálnu od veľkosti vstupu, a teda každé z nich sa zmestí do pracovnej premennej.

- Vaše programy nesmú používať rekurziu.

Príklad 1. Ukážeme si log-space program, ktorý nájde maximum v poli čísel.

Listing programu (Pascal)

```
var n: integer;                  { vstupná premenná: počet čísel }
    A: array [1..n] of integer;  { vstupná premenná: pole čísel }
    m: integer;                  { výstupná premenná: pozícia maxima }
    i, j: integer;               { pracovné premenné }
begin
    j := 1;
    for i := 2 to n do
        if A[i] > A[j] then j := i;
    m := j;
end;
```

Jediné dve premenné sú `i` a `j` a zjavne nadobúdajú len hodnoty z rozsahu $1 \ldots n$. Tým sú teda splnené všetky potrebné podmienky a teda skutočne ide o log-space program.

Príklad 2. Nasledujúci log-space program nájde v poli čísel to, ktoré sa tam vyskytuje najčastejšie.

Listing programu (Pascal)

```
var n: integer;                  { vstupná premenná: počet čísel }
    A: array [1..n] of integer;  { vstupná premenná: pole čísel }
    m: integer;                  { výstupná premenná: jedna pozícia najčastejšieho }
    i, j, c, cmax: integer;      { pracovné premenné }
begin
    cmax := 0;
    for i := 1 to n do begin
        { spočítame, koľkokrát sa vyskytuje A[i] }
        c := 0;
        for j := 1 to n do
            if A[j] = A[i] then c := c+1;
        { je to viac ako doterajšie maximum? }
        if c > cmax then begin
            cmax := c;
            m := c;
        end;
    end;
end;
```

Opäť si ľahko rozmyslíme, že hodnoty pracovných premenných nikdy neprekročia n. Taktiež všetky ostatné požiadavky na log-space program sú dodržané.

Prečo práve takéto programy?:

Isto ste si už položili otázku, prečo takéto programy nazývame log-space a načo je vlastne dobré sa nimi zaoberať.

Najpresnejší spôsob merania pamäťovej zložitosti daného programu dostaneme vtedy, keď zistíme počet *bitov pamäte*, ktorú potrebuje v závislosti od veľkosti vstupu.

Existuje síce pár veľmi jednoduchých problémov, ktoré vieme riešiť napríklad len s troma bitmi pamäte, veľa ich ale nie je a nebudú nás príliš zaujímať. My sa sústredíme na programy, ktoré vstupné dáta dostanú v nejakom n-prvkovom poli. Na to, aby sme vôbec mohli k prvkom takéhoto poľa rozumne pristupovať, potrebujeme do neho vedieť *indexovať*. A na to treba mať premennú, ktorá môže nadobudnúť n rôznych hodnôt.

Jeden bit pamäte má dva rôzne stavy: 0 alebo 1. Ak máme k-bitovú premennú, tá môže nadobúdať 2^k rôznych stavov – nezávisle pre každý z k bitov máme dve možnosti. Tieto stavy si my potom rôzne interpretujeme: raz ako znaky, inokedy ako čísla, a podobne. Napríklad taká 8-bitová premenná môže mať 256 rôznych stavov. Niekedy tieto stavy môžeme považovať za čísla od 0 do 255, inokedy za čísla od -100 do 155, a ešte inokedy za znaky v 8-bitovom ASCII kódovaní.

A tú istú úvahu môžeme spraviť aj opačne. Ak potrebujeme premennú, ktorá vie mať n rôznych hodnôt, potrebujeme, aby pre jej počet bitov k platila nerovnosť $2^k \geq n$. Inými slovami, najmenšie vyhovujúce k je $\lceil \log_2 n \rceil$ (t.j. horná celá časť dvojkového logaritmu čísla n).

Dostávame teda nasledovný záver: na prácu s n-prvkovým poľom určite treba aspoň rádovo $\log_2 n$ bitov pamäte. Toto je teda v istom zmysle najmenšia prakticky zaujímavá pamäťová zložitosť programov.

No a práve takúto (asymptotickú) pamäťovú zložitosť budú mať aj všetky programy, ktoré budete písať vo svojich riešeniach. Totiž naše obmedzenie na počet a veľkosť premenných, ktoré smiete používať, zaručuje, že celkový počet bitov pamäte, ktoré použijete, bude nanajvýš rádovo logaritmický od n. Odtiaľ teda pochádza názov „log-space“ – sú to programy, ktoré používajú $O(\log n)$ bitov pamäte.

(Príklad: Ak použijete 3 premenné, z ktorých každá môže nadobúdať hodnoty od 1 po n^5, bude váš program používať $3\lceil \log_2 n^5 \rceil \approx 15 \log_2 n$ bitov pamäte.)

O zákaze používania polí a rekurzie:

Teoreticky by sa do logaritmického počtu bitov pamäte nejaké maličké polia

zmestili – ale chcelo by to, aby súčet veľkostí všetkých prvkov bol nanajvýš logaritmický. Teda napríklad by sme mohli mať pole obsahujúce $2 \log n$ čísel z rozsahu 0 až 7. Alebo pole obsahujúce 3 čísla z rozsahu 1 až n^2. Nič z toho vám zrejme pri súťažných úlohách nepomôže, preto sme pre jednoduchosť polia zakázali úplne. (Namiesto poľa obsahujúceho 3 čísla predsa vždy môžete použiť 3 vhodne pomenované premenné.)

Druhou konštrukciou, ktorú sme museli zakázať, je rekurzia. Treba si totiž uvedomiť, že pamäť počas behu programu nepoužívame len na premenné. Vždy, keď v programe zavoláme nejakú funkciu, musí sa vyrobiť (na tzv. zásobníku) záznam, kde si okrem iného program zapamätá správnu návratovú adresu a tiež hodnoty parametrov, s ktorými sme danú funkciu zavolali. Taktiež lokálne premenné pre danú funkciu niekam treba uložiť. A tu práve rekurzia začína robiť problémy.

Totiž ak máme program pozostávajúci z viacerých funkcií, ale bez použitia rekurzie (t.j. každá funkcia volá len tie, ktoré boli v programe uvedené pred ňou), vždy ho vieme prerobiť („dosadením“ celého tela funkcie namiesto každého jej volania) na program, ktorý počíta to isté, ale už nepoužíva funkčné volania. Môžete si rozmyslieť, že sa pri tejto zmene nijak výrazne nezmenia pamäťové nároky programu.

Akonáhle však povolíme rekurziu (funkciu, ktorá volá sama seba, prípadne viacero funkcií, ktoré sa vedia volať navzájom), mohlo by sa stať, že pri behu programu nám bude postupne vznikať viac a viac lokálnych premenných a celková pamäť ľahko narastie nad logaritmickú. Preto radšej rekurziu úplne zakazujeme. Rovnako ako pri poliach, aj tu by malo platiť, že vám toto obmedzenie nijak zásadne nesťaží riešenie súťažných úloh.

Zadania domáceho kola kategórie B

B-I-1 Veže

Na veľkú šachovnicu ktosi rozostavil množstvo šachových veží. Niektoré boli biele, iné zas čierne. Barborka je vášnivá šachistka. Rada by si spočítala, koľko dvojíc veží sa navzájom ohrozuje. Keďže ich je ale tak veľa, bojí sa, že sa pomýli. Lepšie by bolo, keby ste jej napísali program, ktorý to vypočíta za ňu.

(Dve veže sa ohrozujú, ak sú opačných farieb, stoja v tom istom riadku alebo stĺpci a medzi nimi nestojí žiadna iná veža.)

Formát vstupu:

V prvom riadku vstupu je jedno celé číslo n, udávajúce rozmer šachovnice. Šachovnica má tvar štvorca, jej riadky aj stĺpce sú očíslované od 1 po n. V druhom riadku vstupu je jedno celé číslo v, udávajúce počet veží.

Nasleduje v riadkov, z ktorých každý popisuje jednu vežu. Popis veže pozostáva z troch celých čísel: číslo riadku, číslo stĺpca a jej farba. (Farba 0 predstavuje čiernu vežu, farba 1 bielu.)

Váš program dostane body za správne vyriešenie piatich testovacích vstupov, ktoré sme si vopred pripravili. Rôzne vstupy popisujú rôzne veľké šachovnice s rôznym počtom veží:

	vstup 1	vstup 2	vstup 3	vstup 4	vstup 5
veľkosť šachovnice n	8	50	1000	1000	1 000 000
počet veží v	20	200	1500	100 000	100 000

Formát výstupu:

Vypíšte jeden riadok a v ňom jedno celé číslo: počet dvojíc veží, ktoré sa ohrozujú. (Môžete predpokladať, že toto číslo bude vždy menšie ako milión.)

Príklad:

Nasledujúci vstup popisuje šachovnicu 8×8. Na nej je 5 veží. Ich rozmiestnenie je znázornené napravo pod správnym výstupom.

Biela veža na (2,4) sa ohrozuje s čiernou na (2,8). Biela veža na (2,4) sa tiež ohrozuje s čiernou na (5,4).

Biele veže na (2,2) a (2,4) sa neohrozujú, lebo sú obe tej istej farby.

Biela veža na (2,2) sa neohrozuje s čiernou vežou na (2,8), lebo medzi nimi stojí iná veža.

Vstup

```
8
5
2 2 1
7 6 0
5 4 0
2 4 1
2 8 0
```

Výstup

```
2


........
.1.1...0
........
........
...0....
........
.....0..
........
```

B-I-2 Dosah

Janko sa dostal do tímu, ktorý programuje umelú inteligenciu do novej tankovej strieľačky `Megatank 3000`. Táto strieľačka sa odohráva na mriežke rozdelenej na štvorcové políčka. Niektoré políčka sú voľné, na niektorých sú prekážky. Tank sa v jednom kroku môže pohnúť o jedno políčka vľavo, vpravo, hore alebo dole. Samozrejme, len vtedy, ak na cieľovom políčku nie je prekážka.

Jankovou úlohou je teraz napísať program, ktorý vyhodnocuje výhodnosť pozície tanku. Presnejšie, potrebuje zistiť, na koľko rôznych políčok sa z daného začiatočného políčka vie tank dostať použitím nanajvýš zadaného počtu krokov. Keďže Janko sa v predchádzajúcom zamestnaní živil ako programátor sústruhov a o tejto úlohe nemá ani šajnu, musíte mu pomôcť vy.

Formát vstupu:

V prvom riadku vstupu je číslo t – počet úloh, ktoré musíte vyriešiť.

V prvom riadku úlohy sú vždy tri celé čísla r, s, q – rozmery mriežky a počet otázok. Mriežka má r riadkov očíslovaných od 1 do r a s stĺpcov očíslovaných od 1 do s.

Nasleduje r riadkov vstupu, pričom každý popisuje jeden riadok mriežky. V každom z týchto riadkov je s medzerou oddelených čísel 0 alebo 1. Číslo 0 predstavuje voľné políčko, číslo 1 predstavuje prekážku.

V ďalších q riadkoch vstupu sú jednotlivé otázky. Každá otázka pozostáva z troch celých čísel y, x, k. Čísla y, x označujú riadok a stĺpec začiatočného políčka. (Môžete predpokladať, že toto políčko neobsahuje prekážku.) Číslo k označuje maximálny počet krokov, ktoré môže tank urobiť.

Váš program dostane body za správne vyriešenie piatich vstupných súborov, ktoré sme si vopred pripravili. (Každý súbor obsahuje viacero úloh. Body za vstup dostanete len vtedy, ak správne vyriešite úplne všetky úlohy, ktoré obsahuje.)

Rôzne vstupy popisujú rôzne veľké mriežky s rôznym maximálnym počtom krokov a rôznym počtom otázok:

	vstup 1	vstup 2	vstup 3	vstup 4	vstup 5
maximálny rozmer mriežky	7	500	100	500	2 000
maximálny počet krokov	3	3	5000	20000	1 000 000
celkový počet otázok	47	63	62	73	12

Formát výstupu:

Pre každú otázku z každej úlohy vypíšte jeden riadok s jedným číslom – počtom políčok, na ktoré sa dá dostať zo začiatočného na maximálne k krokov.

Príklad:

Vstup

```
2
4 4 2
0 0 0 0
0 1 1 0
0 0 1 0
0 0 0 0
1 1 1
4 1 3
2 3 1
0 0 1
1 1 1
1 1 10
```

Výstup

```
3
8
2
```

Tento súbor obsahuje dve úlohy. V prvej z nich má mriežka rozmery 4×4*, v druhej* 2×3*.*

V prvej úlohe treba zodpovedať dve otázky. V prvej tank začína na políčku $(1, 1)$ *a smie spraviť 1 krok. V druhej otázke tank začína na políčku* $(4, 1)$ *a smie spraviť 3 kroky. Táto otázka je znázornená na nasledujúcom obrázku. Hviezdičkou sú označené políčka, na ktoré sa tank vie dostať.*

```
* 0 0 0
* 1 1 0
* * 1 0
* * * *
```

B-I-3 Súčet súčtov

Deti v druhej bé zase na hodine matematiky nedávali pozor a ohadzovali sa kriedou. A tak im dal učiteľ za trest úlohu: Na tabuľu napísal postupnosť čísel. Každý žiak dostal pridelený jeden úsek postupnosti, ktorý mal sčítať. Zhodou okolností bolo všetkých možných úsekov presne toľko ako žiakov, takže každému sa ušiel iný úsek. Keď každý žiak sčítal svoj úsek, napísal svoj výsledok na list papiera. Nakoniec museli žiaci zozbierať všetky listy papiera a sčítať čísla na nich. A až keď učiteľovi oznámili celkový výsledok, pustil ich domov.

Ukážeme si to celé na malom príklade. Majme triedu s desiatimi žiakmi. Učiteľ na tabuľu napísal postupnosť $(10, 20, 10, 30)$. Následne rozdelil deťom prácu:

– štyria žiaci dostali sčítať úseky dĺžky 1: (10), (20), (10) a (30),
– traja žiaci dostali sčítať úseky dĺžky 2: $(10, 20)$, $(20, 10)$ a $(10, 30)$,
– dvaja žiaci dostali sčítať úseky dĺžky 3: $(10, 20, 10)$ a $(20, 10, 30)$,
– jeden žiak (ten, čo najviac vyrušoval) dostal sčítať celú postupnosť.

Žiaci takto dostali na svojich papieroch výsledky 10, 20, 10, 30, 30, 30, 40, 40, 60 a 70. Súčtom všetkých týchto výsledkov je číslo 340.

Súťažná úloha:

Napíšte program, ktorý bude túto úlohu vedieť riešiť pre čo najdlhšie postupnosti.

Pri písaní programu môžete predpokladať, že sa vám výsledok zmestí do bežnej číselnej premennej. Netreba sa teda zaoberať pretečením premenných.

Ľubovoľné funkčné riešenie s dobrým popisom (bez ohľadu na časovú zložitosť) môže dostať aspoň štyri body. Riešenie, ktoré dostane plný počet bodov, si hravo poradí aj s postupnosťou tvorenou stotisíc číslami.

Formát vstupu a výstupu:

V prvom riadku vstupu je jedno celé číslo n, udávajúce dĺžku postupnosti na tabulu. V druhom riadku je n malých kladných celých čísel, tvoriacich dotyčnú postupnosť.

Vypíšte jeden riadok a v ňom jedno celé číslo: súčet súčtov všetkých súvislých úsekov danej postupnosti.

Príklad:

Vstup

```
4
10 20 10 30
```

Výstup

```
340
```

Toto je príklad rozpísaný v zadaní.

B-I-4 Paralelné trampoty

Podúloha A – 6 bodov:

Bola raz jedna trieda, v tej triede bola krieda. Tá krieda bola biela a bolo jej dosť veľa. Okrem zásob kriedy na desať rokov bola v triede ešte tabuľa a na tabuli krásna veľká **nula**. Pred triedou stálo 30 detí. Každé z nich postupne 30-krát zopakovalo nasledovný postup:

- Nejaký čas (aký dlhý sa mu chcelo) počkalo.
- Vošlo do triedy, prečítalo si číslo z tabule, zapamätalo si ho a vyšlo zase von.
- S vypätím všetkých síl k zapamätanému číslu pripočítalo 1. (Aj toto mohlo trvať ľubovoľne dlho.)
- Vošlo do triedy, zmazalo tabuľu a napísalo na ňu výsledok svojho výpočtu.

Konkrétne dieťa vždy najskôr dokončí celý postup, až potom začne odznova. Teda sa napríklad nemôže stať, že to isté dieťa dvakrát po sebe príde prečítať tabuľu, musí medzi tým prísť zapísať výsledok.

Na prvý pohľad je očividné, ako to celé dopadne, nie? Na konci bude na tabuli číslo $30 \times 30 = 900$.

Na druhý pohľad to už až také očividné nie je. Samozrejme, 900 je *najväčší* možný výsledok. Mohlo sa však ľahko stať, že skutočný výsledok bude menší. Nás zaujíma opačný extrém. Vašou úlohou je nájsť *čo najmenšiu* hodnotu, ktorá mohla byť na tabuli po tom, ako každé z 30 detí 30-krát zopakovalo popísaný postup.

Počet bodov, ktoré dostanete, bude závisieť od toho, ako malú hodnotu sa vám podarí zostrojiť. Na plný počet bodov je potrebné nájsť *najmenšiu možnú* hodnotu a stručne zdôvodniť, prečo menšiu už nevieme dosiahnuť.

Podúloha B – 4 body:

Na podobnom princípe je založený jeden typ zraniteľnosti počítačových systémov, tzv. *race condition*. Ide o situácie, kedy si programátor systému neuvedomil, že výsledok súbežného vykonávania viacerých procesov môže závisieť od toho, v akom poradí sa jednotlivé kroky vykonajú. Ukážeme si to na jednoduchom príklade. (Na pár miestach používame Linuxovú terminológiu. Neboj sa jej, na riešenie súťažnej úlohy netreba poznať Linux.)

Samko je správcom počítača, na ktorom má účty viacero ďalších užívateľov. Nás budú zaujímať dvaja z nich: maliarka Marienka a hacker Janko. Marienka nakreslila krásneho poníka a uložila ho do súboru `ponik1.png` vo svojom domovskom adresári. Janko by si ho chcel pozrieť, Marienka však nastavila prístupové práva tak, že sa k nemu vie dostať len ona (a samozrejme správca). A tak mal Janko smolu. Až jedného dňa prišiel Samko za Jankom a hovorí: „Pozri, akú kúl vec som spravil. Program `posli`, ktorému napíšeš názov súboru a e-mailovú adresu, a on ti na ňu ten súbor pošle.“ Janko si všimol, že Samkov program beží s právami správcu systému. A viac mu už nebolo treba. Spustil „`posli /home/marienka/ponik1.png jankov@mail`“ a mal poníka.

Samko chybu odhalil a opravil: prirobil do programu kontrolu, že posielaný súbor musí byť v domovskom adresári užívateľa, ktorý ho práve spustil. Ani to však Janka nezastavilo, mal v zásobe ďalší trik: *symbolickú linku*. (Vo Windows je podobným objektom *zástupca*, po anglicky shortcut.) To je súbor, ktorý nezaberá skoro žiadne miesto, len hovorí: „môj obsah je rovnaký ako obsah tohto iného súboru“. A tak si Janko vo svojom adresári vyrobil symbolickú linku `2.png`, ukazujúcu na súbor `/home/marienka/ponik2.png`, ktorý Marienka medzičasom nakreslila. A už to išlo: Janko spustil „`posli /home/janko/2.png jankov@mail`“. Samkov program následne skontroloval, že ide o súbor v adresári `/home/janko`, prečítal ho, poslal mailom a Janko mal už druhého poníka.

A tu sa už dostávame k našej súťažnej úlohe. Medzi časom Marienka nakreslila tretieho poníka a Samko vylepšil bezpečnosť svojho programu. Samkov program teraz funguje nasledovne: keď ho užívateľ X spustí ako „`posli /nejaka/cesta/subor email`“, program postupne:

1. otvorí adresár `/nejaka/cesta` a skontroluje, či X má právo čítať `subor`.
2. ak áno, otvorí `subor`, načíta ho a prepošle ho na adresu `email`.

Toto vylepšenie by hravo zmarilo Jankov predchádzajúci pokus – totiž Janko síce vedel vyrobiť v svojom adresári symbolickú linku ukazujúcu na obrázok poníka, ale obsah takto vytvoreného súboru nemal právo čítať.

Navrhnite, ako má Janko postupovať, aby sa dostal k súboru `ponik3.png`.

Zadania krajského kola kategórie A

A-II-1 Výkrm

Kráľ Hurim a kráľovná Totňa sa vydali na cestu Lineárnym kráľovstvom.

Verné svojmu menu, Lineárne kráľovstvo je tvorené jedinou cestou. Na tej je n miest, ktoré sú postupne (v poradí, v akom na ceste ležia) očíslované od 1 po n. Hurim a Totňa pôjdu krajinou v tom smere, v ktorom sú číslované mestá, pričom sa postupne zastavia v práve ℓ z nich.

V každom meste varia práve jednu miestnu špecialitu. Môže sa stať, že viaceré mestá majú tú istú špecialitu. V celom kráľovstve je rôznych miestnych špecialít presne m a sú očíslované od 1 po m.

Totňa vie, že Hurim má mlsný jazýček, chcela by teda vybrať zastávky tak, aby sa mu podávané pokrmy páčili. Kráľovskí radcovia pracovali dňom i nocou, až nakoniec prišli s k návrhmi Hurimovho stravovania. Každý návrh je postupnosť ℓ špecialít, ktoré keď Hurim v danom poradí zje, bude na vrchole blaha.

Súťažná úloha:

Totňa sa síce potešila, že má na výber hneď k rôznych návrhov, no optimizmus ju rýchlo prešiel. Čo ak sa niektoré (alebo nebodaj všetky) z nich nedajú v jej kráľovstve realizovať? A keďže miest aj plánov je strašne veľa, potrebuje Totňa vašu pomoc.

Napíšte program, ktorý o každom pláne zistí, či existuje taká postupnosť ℓ zastávok v mestách, pri ktorej Hurim zje presne predpísanú postupnosť jedál.

Upozorňujeme, že sa kráľovský pár nesmie pri svojej ceste vracať späť (do už prejdeného mesta) ani sa dvakrát po sebe zastaviť v tom istom meste.

Formát vstupu:

V prvom riadku vstupu je počet miest n, počet špecialít m (pričom $m \leq n$), počet návrhov stravovania k a počet zastávok ℓ.

V druhom riadku je n celých čísel $s_1, \ldots, s_n$. Číslo s_i ($1 \leq s_i \leq m$) je číslo špeciality podávanej v meste i.

Nasleduje k riadkov, každý z nich popisuje jeden návrh Hurimovho stravovania. Presnejšie, každý z týchto riadkov obsahuje postupnosť presne ℓ celých čísel: čísla špecialít, ktoré by mal Hurim počas výletu v danom poradí jesť.

Formát výstupu:

Pre každý návrh vypíšte jeden riadok s textom „`ano`“ ak sa daný návrh dá realizovať, resp. „`nie`“ ak sa realizovať nedá.

Hodnotenie:

Aspoň 9 bodov môže získať každé riešenie, ktoré efektívne vyrieši každý vstup s $n \leq 100\,000$ a $k \cdot \ell \leq 100\,000$.

(O udelení 10 alebo 9 bodov rozhoduje optimálnosť asymptotickej časovej zložitosti riešenia.)

Až 7 bodov môžete získať za riešenie, ktoré navyše predpokladá aj $m \leq 10$.

Až 5 bodov môžete získať za riešenie, ktoré efektívne vyrieši ľubovoľný vstup v ktorom $k \cdot (n + \ell) \leq 1\,000\,000$.

Za ľubovoľné korektné riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 20$, môžete získať 3 body.

Príklad:

Vstup

```
10 4 4 3
1 2 1 3 4 1 2 1 3 4
1 4 2
4 3 2
1 1 1
4 4 4
```

Výstup

```
ano
nie
ano
nie
```

Prvý návrh sa dá realizovať napr. zastávkou v treťom, piatom a následne siedmom meste.
Tretí návrh sa dá realizovať napr. zastávkou v prvom, šiestom a ôsmom meste.

A-II-2 Vyvážené jablone

Na okraji mestečka Ponyville (viď seriál My Little Pony: Friendship is Magic) žije rodina poníkov, ktorá sa živí pestovaním jabĺk. Darí sa im, stromy v ich sade sú jablkami úplne obsypané. Applejack si ale nedávno všimla, že sa im niektoré stromy ohýbajú. Totiž hen rastie jabĺk veľa, tam zase málo, a tak jabloň nie je vyvážená.

Všetky jablone v sade sú *binárne*. Binárna jabloň je strom veľmi špeciálneho tvaru. Skladá sa z uzlov, vetiev, listov a jabĺk. Najspodnejší z uzlov nazývame

koreň. Z každého uzlu vedú dohora práve dve vetvy. Na hornom konci každej vetvy je buď ďalší uzol, alebo list. Celý strom drží pokope (od koreňa sa po vetvách vieme dostať ku každému inému uzlu) a jeho vetvy sa nikde nespájajú (takže sa po nich nikde nedá chodiť do kolečka).

Jablká rastú len pri listoch stromu. Pri každom liste môže rásť ľubovoľne veľa jabĺk (aj nula).

Pre ľubovoľnú vetvu v strome si môžeme všimnúť, že nad touto vetvou leží nejaká ucelená časť stromu. V tejto časti stromu sa môžu nachádzať aj nejaké jablká. Ich celkový počet nazveme *záťažou* dotyčnej vetvy.

Uzol stromu je *vyvážený*, ak platí, že obe vetvy, ktoré z neho vedú dohora, majú presne rovnakú záťaž.

Celý strom je *vyvážený*, ak sú vyvážené úplne všetky jeho uzly. Ak náhodou strom nie je vyvážený, jediný spôsob, ktorým to smieme zmeniť, je ten, že odtrhneme niektoré vhodne zvolené jablká.

Súťažná úloha:

Napíšte program, ktorý zistí, koľko *najmenej* jabĺk treba z binárnej jablone otrhať, aby bola vyvážená.

Formát vstupu a výstupu:

Môžete predpokladať, že vstup je korektný, teda že naozaj popisuje binárnu jabloň. Pre jednoduchosť budeme uzly a listy jablone označovať spoločným názvom *vrcholy*. V prvom riadku vstupu je číslo n, udávajúce celkový počet vrcholov na našej jabloni. Vrcholy si očíslujeme od 1 po n tak, aby vrchol s číslom 1 bol koreň.

Nasledujúcich n riadkov popisuje jednotlivé vrcholy. Ak je vrchol i uzol, tak je v i-tom z týchto riadkov text „`U` x_i y_i“, kde x_i a y_i sú čísla vrcholov, ktorými končia vetve idúce z uzlu i dohora. Ak je vrchol i list, tak je v i-tom z týchto riadkov text „`L` j_i“, kde j_i je počet jabĺk rastúcich pri tomto liste. Pri písaní programu môžete predpokladať, že sa vám celkový počet jabĺk na strome zmestí do bežnej celočíselnej premennej.

Vypíšte jeden riadok a v ňom jedno celé číslo: najmenší celkový počet jabĺk, ktoré keď z vhodných listov odtrhneme, dostaneme vyvážený strom. Všimnite si, že odpoveď vždy existuje – v najhoršom vždy môžeme otrhať úplne všetky jablká, lebo strom bez jabĺk je zjavne vyvážený.

Hodnotenie:

Plných 10 bodov dostanete za riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 1\,000\,000$.

Až 6 bodov môžete získať za riešenie, ktoré je efektívne pre $n \leq 1000$ a najviac 1000 jabĺk na strome.

Aspoň 3 body budú za ľubovoľné riešenie, ktoré je efektívne pre $n \leq 1000$ a najviac 20 jabĺk na strome.

Príklady:

Vstup

```
3
U 2 3
L 14
L 17
```

Výstup

```
3
```

Odtrhneme tri jablká vo vrchole číslo 3. Tým bude vo vrcholoch 2 a 3 po 14 jabĺk a jabloň bude vyvážená.

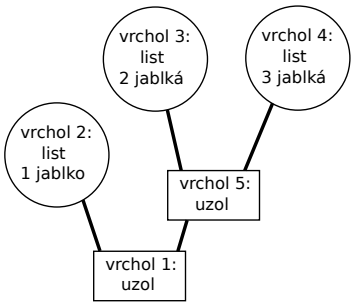
Vstup

```
5
U 2 5
L 1
L 2
L 3
U 3 4
```

V tomto prípade musíme otrhať všetky jablká.

výstup

```
6
```



A-II-3 Zaokrúhľovanie

Janko chcel Peťku prekvapiť, a tak jej na Vianoce daroval krásnu tabuľku plnú reálnych čísel. Lenže čo čert nechcel, Peťke sa páčia len tabuľky plné celých čísel, a také veru v tabuľke nebolo ani jedno. Teraz si to teda musí Janko u nej vyžehliť. Teda pardón, musí si to zaokrúhliť.

Našťastie si Janko všimol, že v Peťkinej tabuľke je súčtom každého riadku aj každého stĺpca celé číslo. Tie sa samozrejme Peťke páčia, a tak ich Janko nesmie zaokrúhľovaním zmeniť.

Každé číslo môže pritom zaokrúhliť buď nadol alebo nahor na najbližšie celé, a to bez ohľadu na hodnotu jeho desatinnej časti. Teda napr. z čísla 6.017 môže vyrobiť buď číslo 6, alebo číslo 7.

Súťažná úloha:

Na vstupe je obdĺžniková tabuľka tvorená $r \times s$ kladnými reálnymi necelými

číslami. Súčet čísel v každom riadku aj stĺpci tejto matice je celé číslo.

Váš program má každé číslo zaokrúhliť nahor alebo nadol tak, aby všetky súčty riadkov a stĺpcov zostali nezmenené, prípadne zistiť, že takýto spôsob zaokrúhlenia neexistuje.

Pri písaní programu môžete predpokladať, že váš programovací jazyk vie dokonale presne ukladať, sčítať, odčítať a porovnávať spracúvané reálne čísla.

Formát vstupu:

V prvom riadku je počet riadkov tabuľky r a počet stĺpcov tabuľky s. Nasleduje r riadkov a v každom s kladných reálnych čísel, z ktorých žiadne nie je celé.

Formát výstupu:

Ak úloha nemá riešenie, vypíšte jeden riadok s textom „`NEDA SA`“.

V opačnom prípade vypíšte r riadkov a v každom s znakov. Znak „`H`“ znamená, že príslušné číslo máme zaokrúhliť nahor, znak „`D`“ zase zaokrúhlenie nadol.

Hodnotenie:

Plných 10 bodov dostanete za riešenie, ktoré dokáže efektívne vyriešiť vstupy, v ktorých $r \cdot s \leq 10^6$. Až 6 bodov môžete získať za riešenie, ktoré dokáže efektívne vyriešiť vstupy, v ktorých $r \cdot s \leq 1000$. Nanajvýš 3 body sú za riešenie, ktoré zvládne vyriešiť vstupy, v ktorých $r \cdot s \leq 20$.

Príklad:

Vstup

```
3 4
5.31 2.39 7.13 0.17
3.33 4.43 1.92 2.32
9.36 1.18 6.95 4.51
```

Súčty riadkov sú 15, 12 a 22.
Súčty stĺpcov sú 18, 8, 16 a 7.

výstup

```
DDHD
HHDD
DDHH
```

Tabuľka po zaokrúhlení:

```
5 2 8 0
4 5 1 2
9 1 7 5
```

A-II-4 Log-space výpočty

Študijný text k tejto úlohe nájdete na strane 11 tejto ročenky.
Pripomíname, že pri riešení súťažných úloh nám **nezáleží** na časovej, ale iba na pamäťovej zložitosti programov.

Súťažná úloha 4a (4 body):

Napíšte log-space program, ktorý vynásobí dve veľké čísla zadané ako postupnosti cifier.

Presnejšie, vstupom je číslo n a dve n-ciferné čísla α a β zadané ako polia $A[0..n-1]$ a $B[0..n-1]$. Číslo $A[i]$ je cifra rádu 10^i v zápise čísla α v desiatkovej sústave. Podobne sú $B[i]$ jednotlivé cifry čísla β. Platí teda:

$$\alpha = A[0] \cdot 10^0 \ + \ A[1] \cdot 10^1 \ + \ \cdots \ + \ A[n-1] \cdot 10^{n-1}$$
$$\beta = B[0] \cdot 10^0 \ + \ B[1] \cdot 10^1 \ + \ \cdots \ + \ B[n-1] \cdot 10^{n-1}$$

Pre všetky prvky polí A a B platí $0 \leq A[i], B[i] \leq 9$, pričom $A[n-1] \neq 0$ a $B[n-1] \neq 0$.

Výstup algoritmu, teda číslo $\gamma = \alpha \cdot \beta$, uložte rovnakým spôsobom po cifrách do poľa $C[0..2n-1]$. Ak je výsledok násobenia menší ako 10^{2n-1}, musí byť $C[2n-1] = 0$.

Príklad: Nech $A = (7, 3, 1)$, teda $A[0] = 7$, $A[1] = 3$ a $A[2] = 1$. Nech $B = (3, 2, 1)$. Tieto dve polia reprezentujú čísla $\alpha = 137$ a $\beta = 123$. Ich súčinom je číslo $\gamma = \alpha \cdot \beta = 16851$. Po skončení programu musí teda platiť $C = (1, 5, 8, 6, 1, 0)$.

Súťažná úloha 4b (6 bodov):

Daný je neorientovaný graf bez cyklov, tzv. *les*. Komponenty súvislosti tohto grafu sú *stromy*. Napíšte log-space program, ktorý na vstupe dostane popis lesa a dva jeho vrcholy a zistí, či dotyčné dva vrcholy ležia na tom istom strome.

Na vstupe dostanete štyri celé čísla. Dve z nich sú počet vrcholov grafu n a počet hrán grafu m. Vrcholy grafu sú označené číslami od 1 do n. Ďalšie dve čísla na vstupe sú čísla u a v dvoch rôznych vrcholov ($1 \leq u < v \leq n$).

Na vstupe tiež dostanete polia $A[1..m]$ a $B[1..m]$. Pre každé i platí, že vrcholy s číslami $A[i]$ a $B[i]$ sú v našom grafe spojené hranou. Môžete predpokladať, že graf na vstupe je les. Platí teda $0 \leq m \leq n-1$ a navyše hrany nášho grafu netvoria žiadny cyklus.

Výstupom programu je celočíselná premenná *spolu*. Do nej priraďte 1, ak vrcholy u a v ležia v tom istom komponente súvislosti, resp. 0, ak nie. Inými slovami, $spolu = 1$ znamená, že sa z vrcholu u dá dostať do vrcholu v tak, že postupne prejdeme po niekoľkých hranách.

Príklad: Nech $n = 5$, $m = 3$, $A = (1, 4, 4)$ a $B = (2, 3, 5)$. Máme teda graf s 5 vrcholmi a 3 hranami: $1 - 2$, $4 - 3$ a $4 - 5$.

Ak by sme vzali $u = 3$ a $v = 5$, správnou odpoveďou je $spolu = 1$, keďže z vrcholu 3 sa dá dostať do vrcholu 5 cez vrchol 4. Naopak, pre $u = 3$ a $v = 1$ má byť $spolu = 0$.

Zadania krajského kola kategórie B

B-II-1 Nutela

Luxusko si kupuje zásoby nutely na dnešný večer. Stojí pred dlhou policou v potravinách, kde sú v rade naukladané nutely v rôznych téglikoch. Tégliky nutely sú na polici zoradené vzostupne podľa veľkosti. Úplne naľavo je najmenšie balenie nutely a napravo najväčšie.

Dnes majú v potravinách akciu: ak si zákazník kúpi práve dve nutely, dostane k nim zadarmo lyžičku. Luxusko už štvrť hodiny stojí a rozmýšľa, ktoré dve nutely si zobrať. Je dôležité, aby sa dobre najedol, takže nemôže kúpiť príliš málo nutely. Ale zase nechce príliš pribrať, takže jej nesmie kúpiť ani priveľa.

Súťažná úloha:

Vašou úlohou je napísať program, ktorý poradí Luxuskovi, ktoré dve nutely má kúpiť, aby súčet ich veľkostí bol aspoň a, ale nie viac než b. Nutely majú byť rôzne – nemôže kúpiť dvakrát tu istú nutelu.

Ak žiadna dvojica nutiel nevyhovuje, vypíšte o tom správu.

Formát vstupu:

V prvom riadku vstupu máte tri kladné celé čísla – počet téglikov na polici n, dolnú hranicu a a hornú hranicu b. Môžete predpokladať, že $a \leq b$.

V druhom riadku je n **rôznych** kladných celých čísel **usporiadaných vzostupne**, pričom i-te z nich vyjadruje veľkosť i-tej nutely na polici.

Veľkosti jednotlivých nutiel aj čísla a a b môžu byť pomerne veľké (napr. rádovo do 10^{18}), pri písaní programu však môžete predpokladať, že sa zmestia do bežných celočíselných premenných.

Formát výstupu:

Ak existujú nejaké dve **rôzne nutely**, ktorých súčet veľkostí je medzi a a b vrátane, vypíšte veľkosti týchto nutiel. V prípade, že je možností viac, vypíšte ľubovoľnú jednu z nich.

Inak vypíšte jeden riadok so správou „nedá sa“.

Hodnotenie:

Plných 10 bodov môžete dostať len za riešenie, ktorého (asymptotická) časová zložitosť je optimálna.

Aspoň 8 bodov môže dostať ľubovoľné riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 100\,000$. Na zisk 6 bodov stačí úlohu efektívne vyriešiť za predpokladov $n \leq 100\,000$ a $a = b$. Na zisk 5 bodov stačí úlohu efektívne vyriešiť za predpokladov $n \leq 100\,000$, $a = b$ a $b \leq 100\,000$.

Ľubovoľné pomalé ale korektné riešenie môže získať až 3 body.

Príklad:

Vstup

```
5 10 12
1 2 4 7 11
```

Výstup

```
4 7
```

Tento vstup má práve dve správne riešenia. Druhým je dvojica 1 + 11.

Vstup

```
4 10 100
3 4 5 123456
```

Výstup

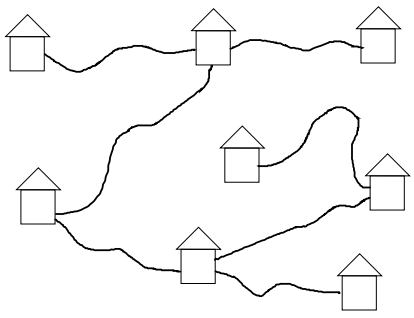
```
nedá sa
```

V tomto vstupe má každá dvojica nutiel buď primalý alebo priveľký súčet.

B-II-2 Pošta

Jano vlastní poštovú spoločnosť J&P. Nedávno sa rozhodol zaviesť novinku: začne ponúkať aj expresné doručovanie. To bude fungovať tak, že človek si priamo do domu zavolá poštára, dá mu list, a poštár ho hneď odnesie na zadanú adresu. Aby Jano vedel, či takáto služba bude komerčne úspešná, rozhodol sa ju najskôr otestovať v malebnej slovenskej dedine Oklieštené Ihličnany.

Oklieštené Ihličnany sú vcelku nezvyčajná dedina. Skladá sa z n domov a $n - 1$ ulíc, pričom každá ulica priamo spája niektoré dva domy. Až na začiatky a konce sa ulice nikde nekrižujú. Po každej ulici sa dá chodiť oboma smermi. Ulice sú navrhnuté tak, aby sa po nich dalo dostať od každého domu ku každému inému. (Odborne povedané, z vyššie uvedených informácií vyplýva, že dedina má stromovú topológiu.)



Príklad dediny a ulíc v nej.

Nová poštová služba sa od prvej chvíle stala veľmi obľúbenou, však v takej dedine je plno noviniek, o ktoré sa chcú jej obyvatelia podeliť. Preto sa stalo, že

hneď prvý deň musel Jano postupne jednu po druhej odniesť $n(n-1)$ zásielok: z každého domu do každého iného domu. Teraz by ho zaujímalo, koľko toho za dnešok s listom v ruke nachodil.

Súťažná úloha:

Na vstupe dostanete popis dediny Oklieštené Ihličnany. Vašou úlohou bude spočítať súčet dĺžok ciest medzi všetkými možnými dvojicami domov. Pod dĺžkou cesty medzi domami x a y rozumieme počet ulíc, ktoré musí Jano prejsť, aby sa z domu x dostal do domu y. (Uvedomte si, že pre každú dvojicu x a y je postupnosť ulíc, ktorými treba ísť, jednoznačne určená, ak nechceme ísť tou istou ulicou tam aj späť.)

Formát vstupu a výstupu:

V prvom riadku vstupu je číslo n určujúce počet domov. Domy sú očíslované od 1 po n. Zvyšok vstupu tvorí $n-1$ riadkov popisujúce ulice. Popis ulice sú dve čísla x, y, ktoré znamenajú, že medzi domami x a y vedie ulica. Je zaručené, že ulice tvoria jeden súvislý strom.

Na výstup vypíšte jedno číslo, súčet dĺžok ciest medzi každými dvoma domami.

Hodnotenie:

Plných 10 bodov dostanete za riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 1\,000\,000$.

Až 7 bodov môžete získať za riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 5\,000$.

Až 4 body môžete získať za riešenie, ktoré efektívne vyrieši ľubovoľný vstup s $n \leq 300$.

Príklad:

Vstup

```
4
1 2
3 1
1 4
```

Výstup

```
9
```

Máme 6 dvojíc domov. Medzi domami 1 a 2 vedie priama ulica, ich vzdialenosť je teda 1. Podobne majú vzdialenosť 1 aj dvojice (1,3) a (1,4). Pre každú z dvojíc (2,3), (2,4) a (3,4) je dĺžka cesty 2. Súčet všetkých týchto vzdialeností je 1+1+1+2+2+2 = 9.

Vstup

```
5
1 2
1 3
2 4
2 5
```

Výstup

```
18
```

Spomedzi 10 dvojíc domov majú štyri dvojice vzdialenosť 1, štyri dvojice vzdialenosť 2 a dve dvojice vzdialenosť 3.

B-II-3 Poklad

Keď Georg jedného dňa vysával dlhočiznú rovnú chodbu vo svojej vile, všimol si, že jeho vysávač zvláštne hrká. Zrazu sa objavil duch jeho (pra)[47]dedka a vyjavil mu, že chodba prechádza miestom, kde pred stovkami rokov zakopali obrovský poklad. Vysávač hrká kvôli tomu, že naň poklad elektromagneticky pôsobí. Duch síce neprezradil, kde presne je pod chodbou poklad ukrytý, ale Georg dostal nápad: zistí to z hrkania vysávača. Čím viac vysávač hrká, tým musí byť poklad bližšie.

Georgov sluch ani pamäť však nestačia na to, aby z hrkania na nejakom mieste priamo povedal, kde sa poklad nachádza. Nanajvýš tak dokáže rozpoznať to, či teraz hrká vysávač viac alebo menej ako na predchádzajúcom vysávanom políčku.

Chodba sa skladá z n políčok uložených v rade za sebou. Políčka sú očíslované zaradom od 1 po n. Poklad leží pod jedným z nich. Georgovi záleží na životnom prostredí a šetrí elektrinou, takže by chcel pri hľadaní pokladu povysávať čo najmenej políčok. (Navyše, čo ak by bol účet za elektrinu vyšší ako cena nájdeného pokladu?)

Súťažná úloha:

Georg chce program, ktorý mu bude radiť, v akom poradí má pri hľadaní pokladu políčka vysávať. Najdôležitejšie je, aby váš program vždy poklad našiel. Navyše sa snažte, aby mal čo najmenšiu spotrebu elektriny – teda aby počet Georgom povysávaných políčok bol (v najhoršom možnom prípade) čo najmenší. Úloha má dve verzie, každú z nich riešte samostatne.

a) (5 bodov) Georg má predlžovačku, vďaka ktorej môže vysávať aj políčka mimo chodby. Inými slovami, číslo vysávaného políčka môže byť ľubovoľné celé číslo. (Poklad je ale určite pod jedným z políčok 1 až n.) Pre dĺžku

chodby platí $2 \leq n \leq 10^9$. Na plný počet bodov nesmie váš program nikdy potrebovať povysávať viac ako 35 políčok.

b) (5 bodov) Georg predlžovačku nemá, a teda môže vysávať iba políčka na chodbe. Inými slovami, číslo každého vysávaného políčka musí byť celé číslo z rozsahu od 1 do n. Plný počet bodov dostanete, ak spotreba vášho programu bude vždy najviac dvojnásobkom najhoršej možnej spotreby optimálneho programu. Teda ak optimálnemu programu vždy stačí povysávať najviac x políčok, vám musí stačiť $2x$.

Formát vstupu a výstupu:

Váš program bude s Georgom komunikovať pomocou štandardného vstupu a výstupu. Na začiatku Georg programu oznámi číslo n, udávajúce počet políčok chodby. Potom bude program bežať v cykle: Na výstup vypíše Georgovi, ktoré políčko má povysávať, a zo vstupu následne načíta Georgovu odpoveď. Tou bude jeden z reťazcov „`blizsie`“, „`dalej`“ alebo „`rovnako`“, podľa toho, či je práve vysávané políčko k pokladu bližšie, od neho ďalej, alebo rovnako ďaleko ako predchádzajúce vysávané políčko. (Pre prvé vysávané políčko mu Georg odpovie „`neviem`“.) Keď už si je program istý polohou pokladu, namiesto vysávania pošle Georga kopať.

Príklad:

Vstup	**Výstup**

`10`
`neviem`
`blizsie`
`rovnako`
	`vysavaj 2`
	`vysavaj 3`
	`vysavaj 9`
	`kop 6`

Udiali sa nasledovné udalosti:

– Program poslal Georga povysávať políčko 2.

– Georg povysával políčko 2 a odpovedal, že o poklade nič nevie.

– Program poslal Georga povysávať políčko 3.

– Georg povysával políčko 3 a odpovedal, že políčko 3 je k pokladu bližšie ako políčko 2.

– Program poslal Georga povysávať políčko 9.

– Georg povysával políčko 9 a odpovedal, že políčko 9 je od pokladu rovnako ďaleko ako políčko 3.

Z toho už program usúdil, že poklad musí byť pod políčkom 6.

B-II-4 Roboti

Na číselnej osi žijú maličkí roboti. Číselnú os majú rozdelenú na políčka dĺžky 1 cm a sami sa pohybujú zásadne s krokom tejto dĺžky. Každým krokom sa teda robot dostane o jedno políčko doľava alebo doprava.

Na každom políčku môže byť (najviac jeden) kamienok. Robot vie kamienky na políčka klásť, zase ich zbierať aj zistiť, či je na políčku kamienok. Kamienky mu v pohybe nijak neprekážajú. Na kladenie ich má pripravených dostatočne veľa, takže môžete predpokladať, že sa mu nikdy neminú. Ak je na číselnej osi viac robotov, nijak si neprekážajú a vidia sa, ak sú na tom istom políčku.

Robotov môžeme programovať. Program pre robota je postupnosť inštrukcií, očíslovaných rastúcimi prirodzenými číslami. Každú inštrukciu robot vykoná za 1 sekundu. Ak sa robotovi minú inštrukcie, sám zastane (ako keby za poslednou inštrukciou bola ešte inštrukcia `koniec`). Roboti poznajú nasledovné inštrukcie:

`koniec:` skonči vykonávanie programu

`nic:` jednu sekundu nerob nič `:-)`

`vlavo, vpravo:` pohni sa o políčko doľava, resp. doprava

`zdvihni, poloz:` zdvihni kamienok, ak tu nejaký je / polož kamienok, ak tu ešte nie je

`pokracuj x:` vo vykonávaní programu pokračuj inštrukciou x.

`ak je kamienok, pokracuj x:` ak je na tvojom políčku kamienok, pokračuj inštrukciou x.

`ak je robot, pokracuj x:` ak je na tvojom políčku iný robot, pokračuj inštrukciou x.

Príklad:

Robot začína na nejakom prázdnom políčku na číselnej osi. Napravo od neho sú na niektorých políčkach rozmiestnené (dokopy aspoň 2) kamienky. Po spustení nasledujúceho programu robot nájde druhý kamienok napravo od neho, zodvihne ho a na danom políčku zastane. (Pre úsporu miesta je program v dvoch stĺpcoch.)

```
10: ak je kamienok, pokracuj 40
20: vpravo
30: pokracuj 10
40: vpravo
```

```
 50: ak je kamienok, pokracuj 999
 60: vpravo
 70: pokracuj 50
999: zdvihni
```

Súťažná úloha:

a) (2 body) Robot začína na nejakom prázdnom políčku na číselnej osi. Aj naľavo, aj napravo od neho je na práve jednom z políčok kamienok. Napíšte program, po ktorého spustení bude robot dokola behať od jedného kamienku k druhému a späť. (Tento program má bežať do nekonečna.)

b) (3 alebo 4 body) Robot začína na nejakom prázdnom políčku na číselnej osi. Aj naľavo, aj napravo od neho je na práve jednom z políčok kamienok. Vzdialenosť medzi kamienkami je párna, teda existuje nejaké políčko p, ktoré je presne uprostred medzi políčkami označenými kamienkom.

   Tri body dostanete za program, po ktorého spustení robot nájde políčko p a zastane na ňom. (Nezáleží na časovej zložitosti programu, ani na tom, kde budú a kde nebudú na číselnej osi kamienky.) Štvrtý bod dostanete za to, ak počas behu vášho programu robot číselnú os uprace: v okamihu, keď na políčku p váš robot ukončí svoj program, nesmie byť na číselnej osi nikde ani jeden kamienok.

c) (4 body) Na číselnej osi nie sú nikde žiadne kamienky. Zobrali sme dvoch robotov a položili ich na dve rôzne políčka. Napíšte *jeden program*, ktorý nahráme do *oboch robotov* a naraz spustíme. Cieľom programu je, aby sa obaja roboti stretli na jednom políčku (je jedno, ktorom) a ukončili tam svoj program.

   Všimnite si, že pri písaní programu neviete, ako ďaleko od seba roboti začínajú. Zdôrazňujeme, že obaja roboti musia dostať *presne rovnaký* program, ktorý potom budú synchrónne krok za krokom vykonávať.

Zadania celoštátneho kola kategórie A

A-III-1 Vodovody

Kráľ Hurim bol veľmi spokojný s cestou po Lineárnom kráľovstve – vďaka vašim radám sa dobre napapkal a pribral 4.7 kg. Následne sa rozhodol, že za odmenu všetkým obyvateľom zavedie pitnú vodu.

Súťažná úloha:

Ako viete, Lineárne kráľovstvo je tvorené jedinou cestou. Na tej je n miest, ktoré sú postupne (v poradí, v akom na ceste ležia) očíslované od 1 po n.

Každé mesto i má svoju produkciu vody p_i. Kladná produkcia znamená, že vodné zdroje v meste produkujú viac vody ako mesto stihne spotrebovať. Samozrejme, niektoré mestá môžu mať produkciu zápornú – nemajú dosť pitnej vody.

Ak máme v nejakom meste prebytok vody, môžeme túto prepraviť potrubiami do susedných miest. Každé potrubie musí spájať dve bezprostredne susediace mestá. Pre každú dvojicu miest $(i, i+1)$ poznáme cenu c_i postavenia potrubia medzi nimi. Voda môže postupne putovať viacerými nadväzujúcimi potrubiami a ľubovoľne sa deliť medzi mestá. Kapacitu potrubí považujte za nekonečnú.

Vašou úlohou je nájsť *najlacnejšiu* množinu potrubí takú, že keď ich postavíme, každé mesto dostane dostatok vody. Teda pre každú skupinu miest prepojených potrubiami musí platiť, že súčet produkcie miest v nej je nezáporný.

Formát vstupu a výstupu:

V prvom riadku vstupu je počet miest n.

V druhom riadku je n celých čísel $p_1, \ldots, p_n$: pre každé mesto jeho produkcia. Súčet všetkých produkcií bude vždy nezáporný. (Vždy teda bude existovať nejaké prípustné riešenie.)

V treťom riadku je $n-1$ kladných celých čísel $c_1, \ldots, c_{n-1}$: pre každú dvojicu susedných miest cena ich prepojenia.

Môžete predpokladať, že všetky hodnoty p_i a c_i aj ich súčty sa pohodlne zmestia do bežných celočíselných premenných. (Formálne môžete napr. predpokladať, že $\sum_i |p_i|$ aj $\sum_i c_i$ sú menšie ako 10^{18}.)

Na výstup vypíšte jediný riadok a v ňom najmenšiu celkovú cenu, ktorú treba zaplatiť za postavenie potrubí.

Hodnotenie:

- Najviac 2 body môžete získať za riešenie, ktoré efektívne vyrieši vstupy s $n \leq 20$.
- Najviac 6 bodov môžete získať za riešenie, ktoré efektívne vyrieši vstupy s $n \leq 1000$.
- Plných 10 bodov môžete získať za riešenie, ktoré efektívne vyrieši vstupy s $n \leq 1\,000\,000$.

Príklad:

Vstup	Výstup
5 -2 10 -2 -5 5 2 2 3 4	7

Najlacnejšie je postaviť prvé tri potrubia, čím prepojíme mestá 1, 2, 3 a 4. Prebytok v meste 2 prerozdelíme ostatným trom mestám (a ešte nám trocha ostane). Mesto 5, ktoré ostalo izolované, má vody dosť. Cena tohto riešenia je 2+2+3 = 7.

Iným korektným (ale drahším) riešením je postaviť potrubia 1-2, 2-3 a 4-5.

A-III-2 Dievka na vydaj

Princezná Baška už je rúča deva. V poslednom čase sa ale zbláznila do seriálu My Little Pony. To sa samozrejme u dospelej následníčky trónu nehodí – ale s Baškou nebola reč. Až keď nezabrali prosby, sľuby ani vyhrážky, pochopil jej kráľovský otec, že tento problém má jediné riešenie: nájdu jej vhodného ženícha, vydajú ju za neho – no a nech si tie prekliate poníky rieši on!

Ako áno, ako nie, keď sa rozkríklo, že budú Bašku vydávať, prihlásilo sa obrovské, prakticky nekonečné množstvo pytačov. Kráľ by chcel Baške vybrať nejakého slušného manžela, nech dievka nehladuje. Je si ale vedomý, že medzi najbohatších pytačov by tiež patriť nemusel – tí sú totiž vplyvní, majú veľké

armády a bohvie, čo by spravili po odhalení Baškinej mánie. Kráľ sa preto rozhodol, že Bašku vydá za k-teho najbohatšieho pytača.

Je známe, že všetci pytači majú navzájom rôzne bohatstvo, takže k-ty najbohatší pytač je jednoznačne určený. Ktorý to ale je? Ako ho v tom dave nájsť? Kráľ zavolal svojich n radcov (pričom n je omnoho menšie ako k), všetkých pytačov rozdelil do n skupín a každú skupinu dal na starosť jednému z radcov.

Každý radca si o pytačoch zo svojej skupiny zistil, kto je ako bohatý, a usporiadal svoju skupinu podľa bohatstva, začínajúc najbohatším pytačom v nej. Bohužiaľ, v tejto situácii ešte stále nebolo jasné, ako nájsť k-teho najbohatšieho zo všetkých pytačov. A čo je horšie, kráľ už nemá žiadnych ďalších radcov, ktorí by mu s tým pomohli. Bude to teda na vás.

Súťažná úloha:

Napíšte program, ktorý bude fungovať nasledovne:

Na začiatku načíta počet radcov n a číslo k udávajúce, koľkého najbohatšieho pytača hľadáme. Radcovia sú očíslovaní od 1 po n. V skupine každého radcu sú pytači očíslovaní vzostupne začínajúc od 1 (pričom číslo 1 má najbohatší v skupine). V každej skupine je aspoň k pytačov.

Následne bude váš program klásť otázky radcom. Otázku položíte zavolaním funkcie `bohatsi(r1,i1,r2,i2)`. Tá v konštantnom čase vráti `true` (pravda) alebo `false` (nepravda) podľa toho, či je i_1-ty pytač v skupine radcu r_1 bohatší ako i_2-ty pytač v skupine radcu r_2.

Keď už si je váš program istý, vypíše, ktorý pytač je ten, ktorého hľadáme, a skončí. Presnejšie, váš program má vypísať dve čísla r a i: číslo radcu a poradové číslo pytača v jeho skupine.

Hodnotenie:

- Najviac 2 body môžete získať za riešenie, ktoré efektívne vyrieši vstupy s $n = 2$ a $k \leq 100\,000$.
- Najviac 6 bodov môžete získať za riešenie, ktoré efektívne vyrieši vstupy s $n \leq 10\,000$ a $k \leq 100\,000$.
- Podľa konkrétnej časovej zložitosti môžete získať 7 až 10 bodov za riešenie, ktoré efektívne vyrieši vstupy s $k \leq 10^{18}$ a čo najväčším n.

Príklad:

Majme $n = 3$ skupiny pytačov a nech $k = 3$, teda hľadáme tretieho najbohatšieho medzi nimi. Pre názornosť si ukážeme aj konkrétne skupiny pytačov a

ich bohatstvo v eurách. (Tieto hodnoty poznajú radcovia, váš program k nim ale nemá prístup.)

skupina 1: 20, 15, 10, ...
skupina 2: 14, 7, 4, ...
skupina 3: 18, 16, 8, ...

Ak by sme v našom programe zavolali funkciu `bohatsi(1,2,3,3)`, dostali by sme odpoveď `true`. Totiž u radcu 1 má pytač 2 bohatstvo 15, u radcu 3 má pytač 3 bohatstvo 8, no a $15 > 8$.

Ak by sme zavolali funkciu `bohatsi(2,3,3,1)`, dostali by sme odpoveď `false`, lebo $4 < 18$.

Váš program by pre tento vstup postupnými volaniami funkcie `bohatsi` mal zistiť, že tretím najbohatším zo všetkých pytačov je v skupine radcu 3 pytač číslo 2 (ten s majetkom 16). Správnym výstupom programu by teda boli čísla „`3 2`“.

A-III-3 Log-space výpočty

Študijný text k tejto úlohe nájdete na strane 11 tejto ročenky.
Pripomíname, že pri riešení súťažných úloh nám **nezáleží** na časovej, ale iba na pamäťovej zložitosti programov.
Nezabudnite ako súčasť riešenia **zdôvodniť**, že vami navrhnutý program je naozaj log-space.

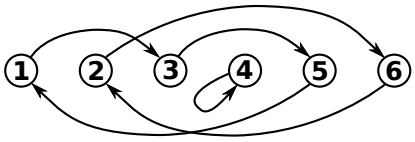
Podúloha A (4 body):

Permutáciou čísel 1 až n voláme takú postupnosť $P[1], \ldots, P[n]$, v ktorej sa každé z čísel 1 až n vyskytuje práve raz. Na permutáciu sa môžeme dívať ako na orientovaný graf, ktorého vrcholmi sú čísla 1 až n a v ktorom máme pre každé i orientovanú hranu z vrcholu i do vrcholu $P[i]$.

Z každého vrcholu teda *vychádza* práve jedna hrana. No a keďže všetky hodnoty $P[i]$ sú navzájom rôzne, znamená to, že do každého vrcholu *vchádza* práve jedna hrana. Náš graf sa teda nutne skladá z niekoľkých *cyklov*. (Niektoré z týchto cyklov môžu byť *slučky* tvorené jedinou hranou z vrcholu i rovno do neho samého.)

Napíšte log-space program, ktorý pre danú permutáciu zistí, koľko má cyklov. Vstupom je celočíselná premenná n a pole $P[1..n]$. Výstupom je celočíselná premenná c.

Príklad. Nech $n = 6$ a nech $P[1..6] = (3, 6, 5, 4, 1, 2)$. Táto permutácia má tri cykly: $(1, 3, 5)$, $(2, 6)$ a (4). Správnym výstupom je teda $c = 3$. Na nasledujúcom obrázku je graf zodpovedajúci tejto permutácii.



Podúloha B (6 bodov):

Strom je súvislý neorientovaný graf neobsahujúci kružnicu. Napíšte log-space program, ktorý pre zadaný strom a dva jeho vrcholy u a v vypočíta vzdialenosť dotyčných dvoch vrcholov – teda najmenšie číslo k také, že na to, aby sme sa dostali z u do v, stačí postupne prejsť po k hranách daného stromu.

Vstup tvoria tri celočíselné premenné (n, u, v) a dve polia (A, B). Premenná n udáva počet vrcholov stromu. Tie sú očíslované od 1 po n. Premenné u a v obsahujú dve rôzne čísla vrcholov.

Strom s n vrcholmi má presne $n - 1$ hrán. Tieto popisujú polia $A[1..n - 1]$ a $B[1..n - 1]$: pre každé i máme v strome hranu spájajúcu vrcholy $A[i]$ a $B[i]$.

Výstup tvorí jediná celočíselná premenná d, do ktorej máte uložiť vzdialenosť vrcholov u a v.

Príklad. Nech $n = 5$, teda máme strom s 5 vrcholmi a 4 hranami. Nech ďalej $A[1 \ldots 4] = (1, 4, 3, 3)$ a $B[1 \ldots 4] = (4, 2, 4, 5)$, teda v našom strome máme hrany 1-4, 4-2, 3-4 a 3-5.

Pre $u = 3$ a $v = 5$ by bolo správnym výstupom $d = 1$, keďže medzi vrcholmi 3 a 5 máme priamu hranu.

Pre $u = 5$ a $v = 1$ by bolo správnym výstupom $d = 3$. Najkratšia cesta z vrcholu 5 do vrcholu 1 vedie cez vrcholy 3 a 4.

A-III-4 Špiónske voľby

Špiónska sieť v Absurdistane síce (v domácom kole OI) utrpela citeľné straty, medzi časom sa však opäť rozvinula do pôvodných rozmerov. Pripomeňme si, že kvôli lepšiemu utajeniu je táto sieť špiónov *riedka*: pre každú podmnožinu špiónov platí, že ak označíme jej veľkosť k, tak v nej bude existovať nanajvýš $3k$ dvojíc špiónov, ktorí sa poznajú. (Poznanie sa je vždy vzájomné.)

Onedlho majú prebehnúť voľby kráľa špiónov. Vo voľbách je päť kandidátov. Pre väčšie utajenie sú označení číslami 0, 1, 2, 3 a 4. Špiónov je n a majú čísla od 0 po $n-1$, vrátane.

Špiónske voľby sa vyhodnocujú inak ako tradičné. Najskôr si samozrejme každý špión vyberie práve jedného kandidáta, ktorého podporuje. Následne každý kandidát dostane toľko bodov, koľko existuje dvojíc špiónov takých, že sa poznajú a obaja daného kandidáta podporujú. Voľby samozrejme vyhrá ten kandidát, ktorý dostane najviac bodov.

Príklad. Majme štyroch špiónov: Pankráca, Serváca, Bonifáca a Medarda. Pankrác, Servác a Bonifác sa všetci navzájom poznajú. Navyše sa ešte poznajú Bonifác s Medardom.

Nech Pankrác a Servác volia kandidáta 0, zatiaľ čo Bonifác a Medard volia kandidáta 1. V tejto situácii by mal kandidát 0 jeden bod (za dvojicu Pankrác-Servác) a kandidát 1 tiež jeden bod (za dvojicu Bonifác-Medard).

Nech teraz Pankrác a Medard volia kandidáta 0, zatiaľ čo Servác a Bonifác volia kandidáta 1. V tejto situácii by mal kandidát 1 jeden bod, ale kandidát 0 by nemal žiaden (lebo Pankrác a Medard sa nepoznajú).

Ak by všetci štyria naši špióni volili toho istého kandidáta, ten by dostal štyri body.

Súťažná úloha:

Prebieha predvolebná kampaň a špióni často menia názor na to, koho budú voliť. Vašou úlohou je napísať knižnicu, ktorá bude sledovať tieto zmeny názoru a priebežne informovať o počte bodov, ktorý majú jednotliví kandidáti.

Sieť špiónov sa počas behu vášho programu meniť nebude. Teda na začiatku sa dozviete, ktoré dvojice špiónov sa navzájom poznajú, a nik z nich počas behu programu nikoho nového nespozná.

Formát implementácie:

Implementujete a odovzdávate (buď v C++ alebo v Pascale) knižnicu obsahujúcu tri funkcie: `spioni`, `zmen_nazor` a `pocet_bodov`.

Pri testovaní vašu knižnicu zlinkujeme s testovacím programom, ktorý sme pripravili my. Testovací program najskôr *jedenkrát* zavolá vašu funkciu `spioni`, čím vašej knižnici oznámi, ako vyzerá sieť špiónov a kto z nich na začiatku volí ktorého kandidáta. (Význam parametrov je popísaný nižšie.)

Následne bude testovací program *veľakrát v ľubovoľnom poradí* volať zvyšné dve funkcie. Volaním funkcie `zmen_nazor(s,v)` vašej knižnici náš program oznámi,

že špión `s` odteraz volí kandidáta `v`. Pri volaní funkcie `pocet_bodov(p)` má vaša funkcia do poľa `p` vyplniť aktuálne počty bodov všetkých piatich kandidátov.

Hlavičky vašich funkcií v C++:

```
void spioni (int n, int m, int dvojice[][2], int voli[]);
void zmen_nazor (int spion, int voli);
void pocet_bodov (int pocet[5]);
```

Pomocné deklarácie a hlavičky vašich funkcií v Pascale:

```
type dvojica = array[0..1] of longint;
type vysledky = array[0..4] of longint;

procedure spioni (n, m : longint; dvojice : array of dvojica;
                  voli: array of longint);
procedure zmen_nazor (spion, voli : longint);
procedure pocet_bodov (var pocet : vysledky);
```

Obmedzenia:

- Pri volaní funkcie `spioni` bude platiť:
  - `n` je celkový počet špiónov (očíslovaných od 0 do $n-1$)
  - `m` je počet dvojíc, ktoré sa poznajú, platí $0 \leq m \leq 3n$ (lebo viac ich byť nevie)
  - `dvojice` je zoznam dvojíc, ktoré sa poznajú: pre každé i od 0 po $m-1$ platí, že sa poznajú špióni `dvojice[i][0]` a `dvojice[i][1]`
  - `voli` udáva kto koho na začiatku volí: pre každé i od 0 po $n-1$ špión i volí kandidáta `voli[i]`.
- Celkový počet volaní funkcií `zmen_nazor` a `pocet_bodov` neprevýši 10^6.
- V testovacích sadách 1, 2 a 3 platí $n \leq 100$.
- V testovacích sadách 4 a 5 platí $n \leq 100\,000$ a funkciu `pocet_bodov` náš program zavolá len raz.
- V testovacích sadách 6 až 10 platí $n \leq 100\,000$ a navyše platí nasledovná podmienka: Nech a a b je ľubovoľná dvojica špiónov, ktorí sa poznajú. Potom aspoň jeden zo špiónov a a b pozná najviac 5 iných špiónov (teda dokopy ich pozná najviac 6).
- V testovacích sadách 11 až 15 platí $n \leq 100\,000$.

Písanie a testovanie riešení:

Dostanete od nás súbor `spioni.cc`, resp. `spioni.pas`. Tento súbor obsahuje kostru knižnice, ktorú máte implementovať. Vo vašom riešení stačí do neho doplniť implementácie funkcií `spioni`, `zmen_nazor` a `pocet_bodov`. Môžete si samozrejme definovať ďalšie pomocné funkcie a lokálne premenné. (V C++ odporúčame všetky globálne premenné deklarovať ako `static`.)

Ďalej dostanete od nás súbor `Makefile` a súbor `main.cc`, resp. `main.pas`. Súbor `main.*` obsahuje ukážkový hlavný program, ktorý môžete použiť na testovanie vášho riešenia. Súbor `Makefile` obsahuje inštrukcie na ich skompilovanie. Stačí na príkazovom riadku napísať `make` a spustia sa správne príkazy, ktoré, ak všetko dobre zafunguje, vyrobia spustiteľný súbor `main`.

Tento súbor očakáva na štandardnom vstupe najskôr celé čísla n a m, potom zoznam m dvojíc čísel špiónov, ktorí sa poznajú, potom zoznam n čísel kandidátov, ktorých na začiatku volia jednotliví špióni. S týmito parametrami zavolá náš program vašu funkciu `spioni`.

Následne môžete na štandardný vstup písať medzerami oddelené príkazy tvaru „`N s v`“ a „`P`“. Po prečítaní príkazu prvého typu náš program zavolá funkciu `zmen_nazor(s,v)`, po prečítaní príkazu druhého typu náš program zavolá funkciu `pocet_bodov` a vypíše na štandardný výstup jeden riadok obsahujúci vrátené hodnoty.

Príklad:

Vstup

```
4 4
0 1   1 2   2 0   2 3
0 0 1 0
P
N 2 0
P
N 3 1
P
N 2 1
P
```

Výstup

```
pocet = 1, 0, 0, 0, 0
pocet = 4, 0, 0, 0, 0
pocet = 3, 0, 0, 0, 0
pocet = 1, 1, 0, 0, 0
```

Vľavo je príklad údajov, ktoré by ste mohli poslať na štandardný vstup skompilovaného programu.

Sieť špiónov je tá z príkladu v zadaní (0 je Pankrác, 1 Servác, 2 Bonifác, 3 Medard). Na začiatku sa zavolá vaša funkcia `spioni` *s popisom tejto siete špiónov a informáciou, že na začiatku Bonifác volí kandidáta 1 a ostatní volia kandidáta 0.*

Následne sa 4× zavolá vaša funkcia `pocet_bodov` *a medzi jednotlivými jej volaniami sa zavolá* `zmen_nazor(2,0)`, `zmen_nazor(3,1)` *a* `zmen_nazor(2,1)`. *Teda postupne sa Bonifác rozhodne voliť kandidáta 0, Medard kandidáta 1, a aj Bonifác kandidáta 1.*

A-III-5 Uhlopriečky

Usamec a Maru majú obaja voľnú hodinu, a tak si dlhú chvíľu krátia zábavnou hrou. Aby nedošlo k nejakým nedorozumeniam, radšej vám povieme, akou.

Hra začína tým, že Maru zoberie čistý list papiera a čiernou farbou naň nakreslí pravidelný n-uholník. Potom doň, opäť čiernou farbou, dokreslí $n-3$ uhlopriečok. Tie však nesmie kresliť len tak, ako sa jej zachce. Musí dodržať dve pravidlá:

1. Žiadne dve uhlopriečky sa nesmú pretínať. (Rozmyslite si, že takýmto dokreslením uhlopriečok Maru určite rozdelí n-uholník na presne $n-2$ trojuholníkov, nič iné nemôže vzniknúť.)

2. Maru musí vyberať uhlopriečky tak, aby mal každý z dotyčných $n-2$ trojuholníkov aspoň jednu stranu spoločnú s pôvodným n-uholníkom.

Keď Maru dokreslí uhlopriečky, začína sa samotná hra. V tej Usamec a Maru striedavo ťahajú (začína Usamec). Ťah spočíva v tom, že si hráč vyberie čiernu úsečku (buď jednu z uhlopriečok, alebo jednu zo strán n-uholníka) a prefarbí ju na červeno. Hru vyhráva hráč, ktorý ako prvý vytvorí červený trojuholník.

Súťažná úloha:

Vašou úlohou je napísať knižnicu, ktorá bude túto hru hrať namiesto Usamca (prvého hráča). Body dostanete, ak vaša knižnica hru vyhrá. V niektorých hrách bude Maru ťahať náhodne, vo väčšine však bude hrať optimálne – teda ak spravíte chybu, prehráte.

Formát implementácie:

Implementujete a odovzdávate (buď v C++ alebo v Pascale) knižnicu obsahujúcu tri funkcie: `hraci_plan`, `tah_usamca` a `tah_maru`.

Pri testovaní vašu knižnicu zlinkujeme s našim testovacím programom. Testovací program najskôr *jedenkrát* zavolá vašu funkciu `hraci_plan`, čím vašej knižnici oznámi, ako vyzerá hrací plán, ktorý nakreslila Maru – teda sa dozviete počet vrcholov n a zoznam nakreslených uhlopriečok.

Následne bude testovací program hrať proti vašej knižnici vyššie popísanú hru, pričom vaša knižnica hrá za Usamca, zatiaľ čo testovací program hrá za Maru. Testovací program bude teda *dokola volať striedavo* vašu funkciu `tah_usamca` a vašu funkciu `tah_maru`.

Pri každom volaní vašej funkcie `tah_usamca` musí táto vrátiť ťah, ktorý chcete spraviť – teda čísla dvoch vrcholov spojených čiernou hranou, ktorú chcete prefarbiť na červenú. (Na poradí čísel vrcholov nezáleží.) Pri každom volaní vašej funkcie `tah_maru` vám v jej parametri testovací program oznámi dve čísla vrcholov určujúce hranu, ktorú na červeno prefarbila Maru.

Akonáhle jeden z hráčov vytvorí červený trojuholník, testovací program podá príslušnú správu a skončí. Ak sa pri niektorom volaní funkcie `tah_usamca` pokúsite spraviť neplatný ťah, testovací program tiež okamžite skončí a automaticky prehrávate. (Neplatný ťah môže byť použitie neexistujúceho čísla vrcholu, použitie dvoch čísel vrcholov ktoré nie sú spojené vôbec, alebo takých dvoch, ktoré už sú spojené červenou hranou.)

Hlavičky vašich funkcií v C++:

```
void hraci_plan (int n, int uhlopriecky[][2]);
void tah_usamca (int hrana[2]);
void tah_maru   (int hrana[2]);
```

Pomocné deklarácie a hlavičky vašich funkcií v Pascale:

```
type dvojica = array[0..1] of longint;

procedure hraci_plan (n : longint;
                      uhlopriecky : array of dvojica);
procedure tah_usamca (var hrana : dvojica);
procedure tah_maru   (hrana : dvojica);
```

Obmedzenia:

- V každom z použitých testovacích vstupov na začiatku hry existuje vyhrávajúca stratégia pre Usamca.
- Pri volaní funkcie `hraci_plan` bude platiť:
  `n` je počet vrcholov mnohouholníka (platí $n \geq 3$, vrcholy sú očíslované 1 až n po obvode); pre každé i od 0 po $n-4$ sú vrcholy `uhlopriecky[i][0]` a `uhlopriecky[i][1]` spojené hranou
- V testovacích sadách 1 až 3 hrá Maru náhodne: v každom ťahu si vyberie náhodnú čiernu hranu a tú prefarbí na červeno. V jednotlivých sadách platí $n \leq 100$, $n \leq 10\,000$ a $n \leq 100\,000$.
- V testovacích sadách 4 až 15 Maru hrá optimálne – akonáhle spravíte zlý ťah, hru už nevyhráte.
- V testovacích sadách 4 až 6 platí $n \leq 10$.
- V testovacích sadách 7 až 10 platí $n \leq 100\,000$ a navyše majú úplne všetky uhlopriečky, ktoré Maru na začiatku nakreslila, spoločný vrchol.
- V testovacích sadách 11 až 15 platí $n \leq 100\,000$.

Písanie a testovanie riešení:

Dostanete od nás súbor `uhlopriecky.cc`, resp. `uhlopriecky.pas`. Tento súbor obsahuje kostru knižnice, ktorú máte implementovať. Vo vašom riešení stačí do neho doplniť implementácie funkcií `hraci_plan`, `tah_usamca` a `tah_maru`. Môžete si samozrejme definovať ďalšie pomocné funkcie a lokálne premenné. (V C++ odporúčame všetky globálne premenné deklarovať ako `static`.)

Ďalej dostanete od nás súbor `Makefile` a súbor `main.cc`, resp. `main.pas`. Súbor `main.*` obsahuje ukážkový hlavný program, ktorý môžete použiť na testovanie vášho riešenia. Súbor `Makefile` obsahuje inštrukcie na ich skompilovanie. Stačí na príkazovom riadku napísať `make` a spustia sa správne príkazy, ktoré, ak všetko dobre zafunguje, vyrobia spustiteľný súbor `main`.

Tento program očakáva ako parameter názov vstupného súboru. Z toho načíta popis hracieho plánu: najskôr číslo n a následne $n-3$ dvojíc čísel vrcholov spojených uhlopriečkou. Následne bude tento program s vami komunikovať cez štandardný vstup a výstup, čím vám umožní ručne hrať proti vašej knižnici. Vždy najskôr zavolá funkciu `tah_usamca`, jej výstup vypíše na štandardný výstup, zo štandardného vstupu načíta váš ťah (t.j. ťah Maru) a ten volaním funkcie `tah_maru` oznámi vašej knižnici.

Príklad:
Vstupný súbor `vstup.txt` obsahuje nasledovné údaje:

```
6
1 3    3 6    6 4
```

Tomuto zodpovedá hrací plán tvaru šesťuholníka s tromi vyznačenými uhlopriečkami. Keď teraz spustíme „`./main vstup.txt`“, môžeme sa zahrať. Jeden možný priebeh hry:

```
volanie tah_usamca vrátilo:  3 1
užívateľ zadal z klávesnice: 4 6
volanie tah_usamca vrátilo:  3 4
užívateľ zadal z klávesnice: 1 6
volanie tah_usamca vrátilo:  3 6
Usamec vyhral
```

Po ťahu „3 6“ hra končí, keďže vznikol červený trojuholník. (Dokonca dva: 136 aj 346.) Poznamenajme, že ani jeden z hráčov nehral v tomto príklade optimálne.

Riešenia domáceho kola kategórie A

A-I-1 Špióni

Vo všeobecnosti je nájdenie *najväčšej kliky* v grafe NP-úplný problém. To okrem iného znamená, že zrejme neexistuje algoritmus, ktorý to zvládne v polynomiálnom čase. Základom riešenia našej úlohy bude teda analýza toho, čím je naša sieť špiónov špeciálna a ako nám to pomôže nájsť lepšie riešenie.

Pripomeňme si podmienku, ktorú podľa zadania naša sieť spĺňa: Pre ľubovoľné prirodzené číslo k platí: ak ľubovoľným spôsobom vyberieme k špiónov, bude medzi nimi najviac $3k$ dvojíc špiónov, ktorí spolu komunikujú.

Z tejto podmienky priamo vyplýva, že dokopy v celej sieti máme nanajvýš $3n$ dvojíc komunikujúcich špiónov (stačí dosadiť $k = n$). Vieme toho však zistiť omnoho viac.

V prvom rade vieme dokázať, že v našej sieti nemôže existovať klika tvorená ôsmimi (a teda ani viac) špiónmi. Prečo? Keď máme osem špiónov, komunikuje spolu nanajvýš $3 \times 8 = 24$ dvojíc. Ale osem špiónov, to je 28 dvojíc, a teda niektorá dvojica špiónov určite spolu nekomunikuje – preto nemôže ísť o kliku.

V tomto okamihu už teda vieme navrhnúť prvý algoritmus s polynomiálnou časovou zložitosťou: Vyskúšame všetky množiny tvorené 1 až 7 špiónmi a o každej overíme, či tvorí kliku. Tento algoritmus má časovú zložitosť $O(n^7)$ a ak ho nejak nezlepšíme, bude použiteľný len pre veľmi malé vstupy.

Lepšie riešenie:

Až 7 bodov sa dalo získať za riešenie, ktoré zvládne vyriešiť vstupy, v ktorých platí $1 \leq n \leq 1000$ a zároveň celkový počet klík špiónov (všetkých, nie len maximálnych) neprevýši 64 000.

Na takéto riešenie nám stačí vylepšiť to predchádzajúce tak, aby sme zbytočne neskúšali možnosti, ktoré nevedú k riešeniu. Presnejšie, ak už o nejakej množine špiónov zistíme, že kliku netvorí, vieme, že nemá zmysel skúšať ani žiadnu jej nadmnožinu. Ak teda použijeme *backtracking* (prehľadávanie s návratom), pri ktorom postupne prezeráme všetky množiny špiónov, ktoré ešte tvoria kliku, vieme dostať dostatočne dobré riešenie.

Kvadratické riešenie:

Prekvapivo, existujú aj zaručene efektívne riešenia našej úlohy.

Kľúčové pozorovanie sme tu už uviedli, ale teraz ho uvedieme v trochu inej

podobe: V ľubovoľnej skupine k špiónov (či už je to klika alebo nie) musí existovať špión, ktorý komunikuje s nanajvýš šiestimi inými. (Ak by to neplatilo, mali by sme v rámci skupiny viac ako $6k/2 = 3k$ dvojíc komunikujúcich špiónov. Alebo v grafovej terminológii: ak máme nanajvýš $3k$ hrán, tak je súčet stupňov vrcholov nanajvýš $6k$, a teda niektorý vrchol má stupeň nanajvýš 6.)

Vyberme si ľubovoľného takéhoto špióna. Sú dve možnosti: buď je súčasťou najväčšej kliky, alebo nie. Ak je súčasťou najväčšej kliky, tvoria ju on + niektorí spomedzi špiónov, s ktorými komunikuje. Keďže tých je nanajvýš 6, máme nanajvýš $2^6 = 64$ skupín špiónov, ktoré treba skontrolovať. A keďže každú z nich tvorí nanajvýš 7 špiónov, celú túto kontrolu vieme spraviť v konštantnom čase.

A potom nám už ostane len druhý prípad: nájsť najväčšiu kliku, do ktorej náš špión nepatrí. Na to stačí nášho špióna zo siete odstrániť. Dostaneme tým sieť tvorenú $n - 1$ špiónmi. Zjavne aj táto sieť spĺňa podmienku zo zadania, a teda pre ňu môžeme celý postup zopakovať – opäť nájdeme špióna, ktorý má nanajvýš 6 známych, a tak ďalej. Takto pokračujeme, až kým sa nám všetci špióni neminú.

Ako je to s časovou zložitosťou tohto algoritmu?

V prvom rade sme vás trochu zavádzali tvrdením, že pre daných (nanajvýš) 7 špiónov vieme v konštantnom čase overiť, či tvoria kliku. Na to potrebujeme pre každú dvojicu špiónov zistiť, či spolu komunikujú. A toto nevieme triviálne spraviť v konštantnom čase. Zatiaľ sa teda uspokojíme s pomalým priamočiarym riešením: vždy, keď potrebujeme vedieť, či spolu špióni x a y komunikujú, prejdeme zoznam všetkých špiónov, s ktorými komunikuje x, a overíme, či je v ňom y. Takto vieme ľubovoľnú potenciálnu kliku overiť v čase $\Theta(n)$.

Priamočiara implementácia celého algoritmu má potom časovú zložitosť $\Theta(n^2)$. Totiž máme n iterácií (jednu pre každého špióna) a pri každej potrebujeme: najskôr čas $\Theta(n)$ na to, aby sme si našli vhodného špióna, ktorý pozná nanajvýš 6 iných, potom opäť čas $\Theta(n)$ na to, aby sme skontrolovali, ktoré podmnožiny jeho známych tvoria kliky, nakoniec čas $\Theta(n)$ na to, aby sme spracovaného špióna z grafu odstránili.

Riešenie vzorové – takmer optimálne:

Ak chceme lepšie riešenie, potrebujeme všetky tri vyššie popísané kroky robiť efektívnejšie.

Efektívnejšie hľadať ďalšieho špióna na spracovanie je ľahké. Špiónov, ktorých už môžeme spracovať (t.j. tých, čo majú 6 alebo menej známych) si budeme udržiavať vo fronte. No a vždy, keď nejakého špióna spracujeme a odstránime,

skontrolujeme každého jeho známeho, či mu práve neklesol počet známych na 6. Ak áno, pridáme ho do fronty. Takto dosiahneme, že vhodného špióna na spracovanie vždy nájdeme v konštantnom čase.

Zvyšné dva kroky vieme zefektívniť použitím lepších dátových štruktúr. Môžeme si napríklad pre každého špióna jeho známych pamätať ako usporiadanú množinu (vyvažovaný binárny strom, `set` v C++), alebo ako neusporiadanú množinu (hešovacia tabuľka, `unordered_set` v C++). V prvom prípade dostaneme riešenie so zaručenou časovou zložitosťou $O(n \log n)$, v druhom prípade dostaneme *očakávanú* časovú zložitosť $O(n)$.

Toto sú riešenia, aké je vhodné písať na praktickej súťaži – využívajú knižničné dátové štruktúry, vďaka čomu sú stručné a prehľadné, a zároveň sú dostatočne efektívne na zisk plného počtu bodov.

Implementačný detail: prezeranie podmnožín:

V nasledujúcom listingu programu si všimnite elegantný spôsob, akým prechádzame všetky podmnožiny známych práve spracúvaného špióna.

Ak máme z-prvkovú množinu, tá má 2^z rôznych podmnožín. Každú podmnožinu P vieme popísať nejakým z-bitovým reťazcom: pre každý prvok pôvodnej množiny napíšeme 0, ak do P nepatrí, alebo 1, ak do P patrí. Všetky podmnožiny takto priamo zodpovedajú číslam od 0 po $2^z - 1$. Keď teda chceme postupne každú podmnožinu prezrieť a spracovať, stačí nám prejsť v cykle (premennou `subset`) cez všetky tieto čísla. Keď už máme v premennej `subset` číslo konkrétnej podmnožiny, jej prvky ľahko nájdeme pomocou bitových operácií.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
using namespace std;

int N, M;
vector< unordered_set<int> > G; // G[x] je množina špiónov, ktorí komunikujú s x
vector<int> najvacsia_klika;

bool je_klika( const vector<int> &spioni ) {
    // overíme, či daná skupina špiónov tvorí kliku
    for (int x : spioni) for (int y : spioni)
        if (x<y && G[x].count(y)==0) return false;
    return true;
}

int main() {
    // načítame vstup a prečíslujeme špiónov na 0 až n-1
    cin >> N >> M;
    G.resize(N);
    for (int m=0; m<M; ++m) {
```

```
        int x,y; cin >> x >> y; --x; --y;
        G[x].insert(y); G[y].insert(x);
    }

    // do fronty na spracovanie dáme špiónov, ktorí už teraz majú <= 6 známych
    queue<int> Q;
    for (int n=0; n<N; ++n) if (G[n].size()<=6) Q.push(n);

    // postupne po jednom spracúvame špiónov
    while (!Q.empty()) {
        int kto = Q.front(); Q.pop();
        vector<int> kandidati( G[kto].begin(), G[kto].end() );

        // pre každú z 2^Z podmnožín kandidátov zistíme, či je to klika
        int Z = kandidati.size();
        for (int sub=0; sub<(1<<Z); ++sub) {
            vector<int> klika(1,kto);
            for (int z=0; z<Z; ++z) if (sub & 1<<z)
                klika.push_back( kandidati[z] );
            if (klika.size() > najvacsia_klika.size() && je_klika( klika ))
                najvacsia_klika = klika;
        }

        // odstránime práve spracovaného špióna
        // ak tým vznikli noví špióni s <= 6 známymi, pridáme ich do fronty
        for (int x : kandidati) {
            G[x].erase( kto );
            if (G[x].size()==6) Q.push(x);
        }
    }
    // vypíšeme prvky najväčšej nájdenej kliky
    for (int x : najvacsia_klika) cout << (x+1) << endl;
}
```

Optimálne riešenie:

Z teoretického hľadiska je zaujímavá aj otázka, či existuje riešenie tejto súťažnej úlohy, ktoré má aj v najhoršom možnom prípade zaručenú lineárnu časovú zložitosť. Odpoveď na túto otázku je kladná.

Pomôžeme si drobným trikom. Rozdelíme riešenie na dve fázy. V prvej fáze zostrojíme poradie, v ktorom budeme špiónov spracúvať. Zároveň si pre každého špióna zapamätáme len tých jeho (nanajvýš 6) známych, ktorých ešte má v okamihu, keď ho spracúvame. Všimnime si, že takto sme zredukovali množstvo zapamätanej informácie na polovicu: ak je na vstupe dané, že špióni x a y spolu komunikujú, budeme si to pamätať len u toho špióna, ktorého skôr spracujeme.

Táto šikovnejšia reprezentácia vstupu nám potom ponúka presne to, čo potrebujeme: vieme pomocou nej v zaručene konštantnom čase overiť, či spolu nejakí dvaja špióni p a q komunikujú. Na to stačí najskôr prejsť zoznam pamätaný pre špióna p (najviac 6 špiónov) a overiť, či je v ňom q, a následne prejsť zoznam pre špióna q (opäť najviac 6 špiónov) a overiť, či je v ňom p. (Alternatívne, ak si pre každého špióna pamätáme poradie, v ktorom má byť spracovaný, stačí prezrieť len jeden správny zoznam.)

A-I-2 Rozvod elektriny

Našou úlohou je rozdeliť zadanú sieť na súvislé neprekrývajúce sa oblasti. V každej oblasti sa má nachádzať práve jedna elektráreň, ktorá bude zásobovať elektrinou všetky mestá v tejto oblasti. Výkon elektrárne nesmie byť menší ako súčet spotrieb miest, ktoré zásobuje. Navyše nás obmedzuje kapacita vedení medzi jednotlivými lokalitami.

Zatiaľ to vyzerá na poriadne zložitú úlohu; našťastie však sieť elektrických vedení tvorí strom. Za jeho koreň vyhlásme vrchol číslo 1, čiže krajské mesto. Strom zavesme za koreň – získame tak možnosť používať intuitívne pojmy nahor (smerom ku koreňu) a nadol. Z každého vrcholu v (okrem koreňa) vedie práve jedna hrana smerom nahor; nazvime túto hranu otcovská a jej cieľový vrchol nazvime otcom vrcholu v. Ostatných susedov vrcholu v budeme volať synovia. Vrcholy rôzne od koreňa, ktoré nemajú synov, sa nazývajú listy.

Myšlienka vzorového riešenia:

Situácia vo vyšších úrovniach stromu je celkom neprehľadná – nevieme na prvý pohľad určiť, ku ktorej elektrárni treba napojiť konkrétny vrchol. Začnime preto od listov, kde máme menší počet možností:

Vezmime si ľubovoľný list ℓ, v ktorom je nejaké mesto; jeho spotrebu označme d_ℓ. Nech ho pripojíme ku ktorejkoľvek elektrárni, musíme využiť jeho otcovskú hranu. To znamená, že kapacita otcovskej hrany nesmie byť menšia ako d_ℓ, inak riešenie úlohy neexistuje.

Ak je otcom listu ℓ nejaká elektráreň p s výkonom d_p, nemáme inú možnosť ako napojiť ℓ na p (cez p by totiž neprešla elektrina zo žiadnej inej elektrárne). V prípade, že $d_\ell > d_p$, výkon elektrárne nestačí na zásobenie mesta, teda riešenie neexistuje. Inak môžeme odstrániť zo stromu vrchol ℓ a znížiť výkon elektrárne p na $d_p - d_\ell$. Dostaneme tak ekvivalentnú úlohu na $n - 1$ vrcholoch, ktorá je riešiteľná práve vtedy, ak je riešiteľná pôvodná úloha.

Ak je otcom listu ℓ nejaké mesto p so spotrebou d_p, potom mesto ℓ musí byť napojené na tú istú elektráreň ako p. Ekvivalentnú menšiu úlohu teda dostaneme tak, že odstránime zo stromu list ℓ a zvýšime spotrebu mesta p na $d_p + d_\ell$.

Vezmime si ľubovoľný list ℓ, v ktorom je nejaká elektráreň a ktorého otcom je tiež elektráreň. Energiu z ℓ cez otca preniesť nemôžeme, preto jednoducho odstránime zo stromu list ℓ a budeme riešiť menšiu ekvivalentnú úlohu.

Ak sa už nedá použiť žiadne z predchádzajúcich zjednodušení úlohy, v každom liste máme nejakú elektráreň, ktorej otcom je nejaké mesto. Vezmime si

také mesto p, ktorého všetci synovia $\ell_1, \ell_2, \ldots, \ell_k$ sú listy (v popísanom prípade musí aspoň jedno také mesto existovať).

Množstvo energie, ktorým môže elektráreň ℓ_i zásobovať mesto p, je obmedzené nielen jej výkonom, ale aj kapacitou vedenia, ktoré ju spája s p. Pre každú z elektrární $\ell_1, \ell_2, \ldots, \ell_k$ si preto vypočítame množstvo prenositeľnej energie. Zrejme nič nestratíme, ak budeme ďalej brať do úvahy len najväčšiu z týchto hodnôt; označme si ju d_ℓ.

V prípade, že je p koreňom, nemáme inú možnosť, ako napojiť ho na elektráreň v niektorom z jeho synov. Porovnáme preto d_ℓ so spotrebou d_p – riešenie existuje práve vtedy, keď maximálne množstvo prenositeľnej energie zo synovskej elektrárne postačuje na pokrytie spotreby mesta p.

Aj vtedy, keď má mesto p otca, budeme sa snažiť pripojiť p k elektrárni v niektorom z jeho synov. Táto voľba nám, intuitívne povedané, ponechá viac energie vo vyšších častiach stromu. Teda ak $d_\ell \geq d_p$, potom pripojíme p k synovskej elektrárni s najväčšou hodnotou prenositeľnej energie. Listy $\ell_1, \ell_2, \ldots, \ell_k$ odstránime zo stromu, mesto p nahradíme elektrárňou s výkonom $d_\ell - d_p$ (množstvo energie nevyužitej v p) a ďalej budeme riešiť menšiu ekvivalentnú úlohu.

Zostáva rozobrať posledný prípad: $d_\ell < d_p$. Vtedy musí byť mesto p pripojené k elektrárni cez otcovskú hranu. Odstránime listy $\ell_1, \ell_2, \ldots, \ell_k$ a riešime zjednodušenú úlohu.

Postupným odoberaním vrcholov podľa predchádzajúceho rozboru prípadov časom dostaneme strom pozostávajúci z koreňa a niekoľkých listov s elektrárňami. Takúto situáciu sme tiež rozobrali.

Efektívna implementácia:

V prvom rade potrebujeme vedieť rýchlo hľadať listy stromu. Našťastie sú lokality na vstupe očíslované v poradí, v akom boli pripájané k súvislej sieti. To znamená, že vrchol číslo n je listom; keď ho odstránime zo stromu, vrchol číslo $n-1$ bude listom; a tak ďalej.

Vrcholy teda budeme spracúvať od konca. Pre každé mesto v si budeme okrem informácií zo vstupu pamätať ešte hodnotu `ponuka[`v`]`, čo bude najväčšie množstvo energie prenositeľnej z jeho synov, ktorých sme už spracovali. K tomu navyše prislúcha hodnota `od_koho[`v`]` – číslo elektrárne, od ktorej pochádza najlepšia ponuka.

Ak je práve spracúvaným listom elektráreň, skúsime zlepšiť ponuku jej otcovi (samozrejme len v prípade, ak to je mesto).

V prípade, že spracúvame mesto, skúsime využiť najlepšiu ponuku a posunúť ju ďalej otcovi zmenšenú o spotrebovanú energiu. Ak ponuka nestačí na pokrytie spotreby mesta, skúsime sa napojiť cez otcovskú hranu. Ak je otcom elektráreň, znížime jej výkon o spotrebu mesta; ak je otcom mesto, zvýšime jeho spotrebu.

Po spracovaní všetkých vrcholov potrebujeme vypísať čísla elektrární, ktoré sme priradili jednotlivým lokalitám. Na to si postupne vypĺňame pole `riesenie`. Väčšinou vieme určiť správnu hodnotu už pri spracovaní vrcholu. Jediný prípad, keď tomu tak nie je, nastáva pri zlyhaní ponuky a spoľahnutí sa na otcovskú hranu, ktorá vedie do iného mesta. Vtedy totiž vieme len to, že vrcholu priradíme rovnakú elektráreň ako neskôr jeho otcovi. Preto nakoniec ešte raz prejdeme poľom `riesenie` odpredu a podopĺňame chýbajúce hodnoty.

Keďže každý vrchol spracujeme v konštantnom čase, celková časová zložitosť je $O(n)$.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
using namespace std;

int n;
vector<bool> je_mesto;
vector<int> otec;         // otec vrcholu v zakorenenom strome
vector<int> kapacita;     // kapacita vedenia do otca
vector<long long> watty;  // výkon elektrárne / spotreba mesta
vector<int> ponuka;       // najväčšie množstvo energie prenositeľnej od synov...
vector<int> od_koho;      // ... a príslušná elektráreň
vector<int> riesenie;     // priradenie elektrární mestám

bool vyries() {
    for (int i = n - 1; i >= 1; --i)
        if (je_mesto[i]) {
            if (ponuka[i] >= watty[i]) {
                riesenie[i] = od_koho[i];
                if (je_mesto[otec[i]]) {
                    int moja_ponuka = min(kapacita[i], (int) (ponuka[i]-watty[i]));
                    if (moja_ponuka > ponuka[otec[i]]) {
                        ponuka[otec[i]] = moja_ponuka;
                        od_koho[otec[i]] = od_koho[i];
                    }
                }
            } else {
                if (kapacita[i] < watty[i])
                    return false;
                if (je_mesto[otec[i]]) {
                    watty[otec[i]] += watty[i];
                } else {
                    if (watty[otec[i]] < watty[i])
                        return false;
                    riesenie[i] = otec[i];
                    watty[otec[i]] -= watty[i];
                }
            }
        } else {
```

```
            riesenie[i] = i;
            if (je_mesto[otec[i]]) {
                int moja_ponuka = min(kapacita[i], (int) watty[i]);
                if (moja_ponuka > ponuka[otec[i]]) {
                    ponuka[otec[i]] = moja_ponuka;
                    od_koho[otec[i]] = i;
                }
            }
        }
    if (ponuka[0] < watty[0])
        return false;
    riesenie[0] = od_koho[0];
    for (int i = 0; i < n; ++i)
        if (riesenie[i] == -1)
            riesenie[i] = riesenie[otec[i]];
    return true;
}

int main() {
    scanf("%d", &n);
    je_mesto.resize(n); otec.resize(n); kapacita.resize(n); watty.resize(n);
    ponuka.resize(n, -1); od_koho.resize(n); riesenie.resize(n, -1);
    je_mesto[0] = true; otec[0] = -1; kapacita[0] = -1;
    scanf("%lld", &watty[0]);
    for (int i = 1; i < n; ++i) {
        char typ;
        scanf(" %c%d%d%lld", &typ, &otec[i], &kapacita[i], &watty[i]);
        je_mesto[i] = (typ == 'M');
        --otec[i];
    }
    if (vyries()) {
        for (int i = 0; i < n; ++i)
            printf("%d%c", riesenie[i] + 1, (i == n - 1) ? '\n' : ' ');
    } else {
        printf("nemozne\n");
    }
}
```

A-I-3 Húsenice

Škorec Ignác si musí zvoliť správny smer, ktorým z hniezda vyletí. Na výber má samozrejme nekonečne veľa možností, takže neprichádza do úvahy riešenie „vyskúšame všetky možnosti a vyberieme najlepšiu z nich“. Intuitívne sa nám však zdá, že „zmysluplných“ možností na výber až tak veľa nebude. Zväčša bude zrejme jedno, či Ignác poletí pod azimutom 47.004 alebo 47.005. Ako ale toto pozorovanie sformulovať exaktne?

Skúšanie len rozumných možností:

Predstavme si, že si Ignác zvolil nejaký smer. Nech je tento smer taký, že nepoletí ponad koniec žiadnej z húseníc. Ignácovu trasu si teda môžeme predstaviť ako polpriamku pretínajúcu vnútro niektorých úsečiek. Túto polpriamku teraz môžeme do ľubovoľnej strany *otáčať* (teda spojito meniť smer, ktorým

vedie), až kým „nenarazíme“ na koniec niektorej z húseníc. Je zjavné, že počas otáčania sa nijak nezmení množina húseníc, ktoré Ignác vyzbiera. Zmena môže nastať v poslednom okamihu – ale ak nastane, ide o zmenu k lepšiemu. Môže sa totiž stať, že pri našom postupnom menení smeru narazíme na koniec *novej* húsenice, ktorú doteraz Ignác uloviť nevedel.

Čo sme práve ukázali? Nech by si Ignác zvolil akýkoľvek smer letu, my mu ho vieme vždy upraviť tak, aby zozbieral aspoň toľko isto húseníc, a zároveň letel ponad koniec jednej z nich. Inými slovami, medzi optimálnymi riešeniami (ktorých môže byť opäť aj nekonečne veľa) určite existuje také, pri ktorom Ignác letí ponad koniec jednej z húseníc.

Takto už ale dostávame prvé funkčné riešenie. Máme n húseníc, každá má dva konce, dokopy teda potrebujeme skontrolovať $2n$ polpriamok. Pre každú z nich spočítame, koľko húseníc by Ignác vyzbieral, a vyberieme najlepšiu možnosť. Pre konkrétnu polpriamku a konkrétnu húsenicu vieme v konštantnom čase zistiť, či sa pretínajú. (Toto podrobnejšie vysvetlíme nižšie.) Konkrétnu polpriamku teda vieme skontrolovať v čase $\Theta(n)$ a tým pádom je celková časová zložitosť tohto riešenia $\Theta(n^2)$.

Za zapamätanie stojí hlavná myšlienka vyššie popísaného riešenia: nájdeme jednoduché kritérium, ako spomedzi množstva možností vybrať nejakú malú zaujímavú sadu, ktorá určite obsahuje (aspoň jednu) optimálnu možnosť. Podobná technika sa dá pri návrhu efektívnych algoritmov (obzvlášť v oblasti výpočtovej geometrie) použiť veľmi často.

Skúste si napríklad podobnou technikou vyriešiť nasledovnú úlohu: Máme obdĺžnikovú dosku tvorenú bodmi na súradniciach $(0, 0)$ až (x_0, y_0). V doske je $n \leq 50$ bodových dier na súradniciach (x_1, y_1) až (x_n, y_n). Chceme z dosky vyrezať čo najväčší (obsahom) obdĺžnik, ktorý bude mať strany rovnobežné so súradnicovými osami a nebude vo vnútri obsahovať žiadnu z dier.

Priesečník úsečky a polpriamky:

Predstavme si, že sme pre Ignáca zvolili nejakú polpriamku a že sme si vybrali jednu konkrétnu húsenicu, o ktorej chceme zistiť, či ju Ignác uloví. Predĺžme Ignácovu trasu aj húsenicu na priamky. Môžu nastať dva prípady: buď sú tieto dve priamky rovnobežné, alebo sú rôznobežné. Ak sú rovnobežné, podmienka v zadaní nám garantuje, že nie sú totožné, a teda nemajú žiaden spoločný bod. Ak sú rôznobežné, majú práve jeden priesečník.

Oboje vieme vypočítať napríklad riešením sústavy vhodných rovníc. Nech je Ignácova priamka určená bodom (x_1, y_1) a nech má húsenica konce (x_2, y_2) a (x_3, y_3). Potom Ignácovu priamku tvoria body tvaru (sx_1, sy_1) pre $s \in \mathbb{R}$ a

húsenicinu priamku zase body tvaru $\Big(x_2 + t(x_3 - x_2),\, y_2 + t(y_3 - y_2)\Big)$ pre $t \in \mathbb{R}$. (Dokonca vieme aj presnejšie povedať, že Ignácovu polpriamku tvoria práve body s $0 \leq s$ a húsenicinu úsečku body, pre ktoré platí $0 \leq t \leq 1$.)

Priesečník našich dvoch priamok teda spočítame tak, že hľadáme dvojicu s, t, pre ktorú dosadením s do Ignácovho vzorca dostaneme ten istý bod ako dosadením t do vzorca pre húsenicu. Takto dostávame dve lineárne rovnice – jednu pre x-ovú súradnicu, druhú pre y-ovú:

$$sx_1 = x_2 + t(x_3 - x_2)$$
$$sy_1 = y_2 + t(y_3 - y_2)$$

Ak sú priamky rovnobežné a nie sú totožné, táto sústava nemá riešenie. Ak sú rôznobežné, existuje práve jedno riešenie. Keď toto riešenie nájdeme, skontrolujeme ešte, či daný priesečník priamok leží aj na Ignácovej polpriamke, aj na húsenicinej úsečke – teda či s a t spĺňajú vyššie uvedené nerovnosti.

Existuje aj elegantnejšie riešenie, ktoré sa zaobíde bez presného výpočtu súradníc priesečníka. Stačí namiesto bodu (x_1, y_1) zobrať nejaký bod (cx_1, cy_1), ktorý už je dostatočne ďaleko, a potom pomocou vektorových súčinov overiť, či sa pretínajú úsečky $(0, 0) - (cx_1, cy_1)$ a $(x_2, y_2) - (x_3, y_3)$.

Efektívnejšie riešenie:

Vyššie popísané kvadratické riešenie je použiteľné, ak by húseníc bolo pár tisíc. Pre väčšie počty húseníc budeme potrebovať efektívnejšie riešenie. Vylepšíme preto naše kvadratické riešenie pomocou ďalšej štandardnej techniky vo výpočtovej geometrii: *zametania*. Základná myšlienka bude jednoduchá – namiesto toho, aby sme každú zaujímavú polpriamku vyhodnocovali úplne odznova, budeme simulovať postupné otáčanie polpriamky a pri tom si udržiavať množinu úsečiek, ktoré v danej chvíli pretína.

Začať môžeme napríklad tým, že Ignácovu polpriamku umiestnime na kladnú poloos osi x, prejdeme všetky húsenice a zistíme, ktoré teraz pretína.

Následne začneme našu polpriamku otáčať proti smeru hodinových ručičiek. Samozrejme, nie spojito, len budeme postupne prechádzať cez všetky situácie, ktoré by spracúvalo aj predchádzajúce, pomalšie riešenie. Aby sme to vedeli robiť efektívne, usporiadame si všetky konce húseníc do poradia, v ktorom cez ne bude naša polpriamka prechádzať. (Odborne sa tomu hovorí, že konce húseníc *usporiadame polárne*. Ide vlastne o usporiadanie, pri ktorom je kľúčom orientovaný uhol od kladnej poloosi osi x po polpriamku prechádzajúcu cez daný bod.)

Ako teraz vyzerá spracovanie novej polohy polpriamky? Ak ide o „začiatok“ húsenice (teda ak ide o húsenicu, ktorú doteraz naša polpriamka nepretínala), zvýšime si počítadlo húseníc, inak ho znížime. Teda každú novú polpriamku spracujeme v konštantnom čase. Celkom slušné zlepšenie, nie?

Jediné, na čo si treba dať pozor, sú situácie, kedy naraz nastane viacero *udalostí* – teda ak v tom istom okamihu viaceré húsenice „začnú“ a viaceré iné „skončia“. Takéto situácie ale ošetríme ľahko. Stačí pri spracovaní uprednostniť začiatky nových húseníc a až po nich spracovať konce.

Najpomalšou časťou tohto riešenia je samotné triedenie: $2n$ koncov húseníc vieme usporiadať v čase $\Theta(n \log n)$. Zvyšok algoritmu má potom lineárnu časovú zložitosť.

Implementačné detaily:

Geometrické úlohy zvádzajú k použitiu reálnych čísel. Počítač však skutočné reálne čísla používať nevie, vie s nimi narábať len približne. A pri tom vznikajú zaokrúhľovacie chyby, s ktorými potom musíme v riešení počítať. (Nikdy by sme napríklad nemali testovať presnú rovnosť dvoch „počítačových reálnych“ čísel. Namiesto toho treba za rovné považovať napr. hocijaké dve hodnoty, ktoré sa od seba líšia menej ako vhodná tolerancia.)

Ak sa to dá, omnoho lepšie je napísať program tak, aby sme si vystačili len s celými číslami. Potom máme istotu, že všetky výpočty s nimi budú presné. Takto vieme riešiť aj našu súťažnú úlohu. Myšlienku len načrtneme, detaily si čitateľ isto ľahko nájde v listingu programu.

V prvej fáze riešenia potrebujeme zistiť, ktoré úsečky pretínajú kladnú poloos osi x, čo ide ľahko. V druhej fáze riešenia potrebujeme polárne usporiadať konce úsečiek. Dva konkrétne body porovnáme tak, že najskôr porovnáme polrovinu, v ktorej ležia. Ak to nerozhodlo, tak sa pozrieme na znamienko vektorového súčinu vektorov, ktoré zodpovedajú našim bodom. A ak ani toto nerozhodlo (znamienko vyšlo 0), tak „začiatky“ majú byť skôr ako „konce“.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
struct point { int x,y,priorita; };
int ZACIATOK=0, KONIEC=1;

bool pretina_xplus(const point &A, const point &B) {
    if ((A.y>0 && B.y>0) || (A.y<0 && B.y<0)) return false; // tá istá polrovine
    if (A.y==0) return (A.x>=0); if (B.y==0) return (B.x>=0); // jeden na osi x
    // inak stačí zistiť, či je správny uhol medzi nimi < 180
```

```
    if (A.y > 0) return (B.x*A.y - A.x*B.y > 0);
            else return (B.x*A.y - A.x*B.y < 0);
}

bool skor_polarne(const point &A, const point &B) {
    int hA = (A.y>0 || (A.y==0 && A.x>0)), hB = (B.y>0 || (B.y==0 && B.x>0));
    if (hA != hB) return (hA > hB); // ak sú v rôznych polrovinách, je jasno
    int vp = A.x*B.y - B.x*A.y; // inak zistíme vektorový súčin
    if (vp != 0) return (vp > 0); // ak vieme, rozhodneme podľa neho
    return A.priorita < B.priorita; // inak je začiatok skôr ako koniec
}

int main() {
    int N; cin >> N;
    vector<point> body; // všetky konce úsečiek, ktoré treba spracovať
    int pretatych_teraz=0;

    for (int n=0; n<N; ++n) {
        // načítame ďalšiu úsečku
        point A, B; cin >> A.x >> A.y >> B.x >> B.y;
        // vymeníme jej konce tak, aby bol uhol A0B menej ako 180 stupňov
        if (A.x*B.y - B.x*A.y < 0) swap(A,B);
        A.priorita=ZACIATOK; B.priorita=KONIEC;
        // podľa toho, či pretína poloos x+, vložíme body na spracovanie
        if (pretina_xplus(A,B)) {
            ++pretatych_teraz;
            if (A.y!=0) body.push_back(A); // ak začína na x+, už je spracovaný
            body.push_back(B);
        } else {
            body.push_back(A);
            body.push_back(B);
        }
    }
    int pretatych_max = pretatych_teraz;
    point optimalny_smer {1,0,false};

    // polárne usporiadame body
    sort( body.begin(), body.end(), skor_polarne );

    // postupne prechádzame a spracúvame všetky body, potom vypíšeme výsledok
    for (point bod : body) {
        if (bod.priorita==0) ++pretatych_teraz; else --pretatych_teraz;
        if (pretatych_teraz > pretatych_max) {
            pretatych_max = pretatych_teraz;
            optimalny_smer = bod;
        }
    }
    cout << optimalny_smer.x << " " << optimalny_smer.y << "\n";
}
```

A-I-4 Log-space výpočty

Riešenia jednotlivých podúloh si ukážeme v trochu inom poradí – na koniec si necháme jedno možné riešenie podúlohy B, ktoré využíva podúlohu C.

Podúloha A – usporiadanie prvkov:

Zjavne nemôžeme použiť žiaden z klasických efektívnych algoritmov – tie

totiž skoro všetky prvky v poli preusporadúvajú, čo my spraviť nevieme. Na druhej strane, nik nás nenúti mať časovú zložitosť $O(n \log n)$. A toto plne využijeme: naše riešenie bude mať kvadratickú časovú zložitosť.

Najskôr uvažujme trochu jednoduchšie zadanie: nech sú všetky prvky poľa $A[1..n]$, ktoré ideme usporiadať, navzájom rôzne. Pozrime sa na konkrétny prvok $A[i]$. Potrebujeme ho umiestniť niekam do výstupného poľa B, ale kam? Odpoveď je jednoduchá: prejdeme pole A a zistíme, koľko z jeho prvkov je menších ako $A[i]$. Práve tie budú v poli B naľavo od neho, správny index kam umiestniť $A[i]$ je teda o jedno väčší od ich počtu.

Ako toto riešenie upraviť pre prípad, keď pole A môže obsahovať viackrát tú istú hodnotu? U prvkov s rovnakou hodnotou zachovávame ich poradie z poľa A. Keď teda spracúvame prvok $A[i]$ a zaujíma nás, koľko prvkov bude v poli B naľavo od neho, zarátame aj prvky s rovnakou hodnotou na pozíciách menších ako i.

Listing programu (Pascal)

```
var n : integer;                          { vstup: dĺžka postupnosti }
    A : array [1..n] of integer;          { vstup: postupnosť čísel }
    B : array [1..n] of integer;          { výstup: usporiadaná postupnosť }
    i, j, vlavo : integer;

begin
    for i := 1 to n do begin
        vlavo := 0;
        for j := 1 to n do begin
            if (A[j] < A[i]) then inc(vlavo);
            if (A[j] = A[i]) and (j<i) then inc(vlavo);
        end;
        B[ vlavo+1 ] := A[i];
    end;
end.
```

Podúloha B – overenie permutácie:

Aj táto podúloha má ľahké riešenie v kvadratickom čase. Opäť, jednoduchšie by sa riešila, ak by všetky prvky v poli A boli navzájom rôzne. Vtedy by nám stačilo pre každý prvok poľa A prejsť pole B a overiť, či sa v ňom tento prvok tiež nachádza.

Ako si teraz poradíme s poľami, v ktorých sa prvky môžu opakovať? Vyššie uvedený algoritmus môžeme zovšeobecniť nasledovne: Postupne pre každý prvok poľa A zistíme, koľkokrát sa vyskytuje v poli A, koľkokrát v poli B, a tieto dve hodnoty porovnáme. Ak niekedy nájdeme rozdiel, vieme, že sa naše dve polia líšia. A naopak, ak má každý prvok poľa A rovnaký počet výskytov aj v poli B, musí už pole B byť nutne preusporiadaním poľa A. (Uvedomte si tiež, že B

má rovnakú dĺžku ako A, a teda nemôže už obsahovať nič navyše.)

Listing programu (Pascal)

```
var n : integer;                          { vstup: dĺžka postupnosti }
    A : array [1..n] of integer;          { vstup: prvá postupnosť čísel }
    B : array [1..n] of integer;          { vstup: druhá postupnosť čísel }
    i, j, pocetA, pocetB : integer;

begin
    for i := 1 to n do begin
        pocetA := 0;
        for j := 1 to n do if (A[i] = A[j]) then inc(pocetA);
        pocetB := 0;
        for j := 1 to n do if (A[i] = B[j]) then inc(pocetB);
        if (pocetA <> pocetB) then begin
            writeln('nie');
            halt;
        end;
    end;
    writeln('ano');
end.
```

Podúloha C – ako sa zbaviť pomocného poľa:

Podľa zadania máme dva log-space programy $\mathcal{F}$ a $\mathcal{G}$. Program $\mathcal{F}$ dostane vstup v poli $A[1..n]$ a vyrobí z neho výstupné pole $B[1..n]$. Program $\mathcal{G}$ dostane na vstupe pole B, ktoré vyrobil program $\mathcal{F}$, a svoj výstup zapíše do výstupného poľa $C[1..n]$.

Ako ich skombinovať do nového programu $\mathcal{H}$, ktorý vyrobí priamo z poľa A pole C? Odpoveď opäť nie je príliš zložitá: Program $\mathcal{H}$ bude robiť presne to isté ako program $\mathcal{G}$ – až na situácie, v ktorých sa program $\mathcal{G}$ snaží čítať z poľa B. Toto pole samozrejme náš program $\mathcal{H}$ nemá k dispozícii. Poznáme ale program $\mathcal{F}$, ktorý ho celé vie vypočítať!

A teda kedykoľvek, keď program $\mathcal{H}$ potrebuje prečítať nejaké políčko $B[i]$, spustíme ako podprogram znova a znova od začiatku program $\mathcal{F}$. Počas behu $\mathcal{F}$ ignorujeme všetky jeho zápisy do poľa B (ktoré nemáme) okrem jediného – zápisu na to políčko $B[i]$, ktoré nás práve zaujíma. Túto hodnotu si uložíme do pomocnej premennej. Keď podprogram $\mathcal{F}$ dobehne, program $\mathcal{H}$ pokračuje v simulácii programu $\mathcal{G}$, pričom ako hodnotu $B[i]$ použije hodnotu, ktorú sme si zistili a uložili do pomocnej premennej.

Potrebujeme sa ešte zamyslieť nad pamäťovou zložitosťou. Ako je to s ňou? Simulujeme program $\mathcal{G}$, ktorý bol log-space. A počas jeho behu občas spustíme ako podprogram program $\mathcal{F}$, ktorý je tiež log-space, a na uloženie jedného políčka jeho výstupu použijeme jednu pomocnú premennú. V súčte sú teda celkové pamäťové nároky dostatočne malé.

Všimnite si tiež, že by takáto konštrukcia nefungovala, ak by si mohli log-space programy prepisovať vstupné pole (potom by sme nevedeli viackrát spustiť $\mathcal{F}$) alebo ak by mohli čítať výstupné pole (potom by sme počas simulácie $\mathcal{F}$ nemohli zahadzovať tie prvky poľa B, ktoré nás práve nezaujímajú).

Ešte raz podúloha B – overenie permutácie:

Podúlohu B sme už síce vyriešili, bude však poučné vyriešiť ju ešte raz. V podúlohe A sme napísali program, ktorý dané pole usporiada. Ak by sme mali pomocné polia, vedeli by sme podúlohu B vyriešiť ľahko: usporiadame pole A, usporiadame pole B a výsledky porovnáme.

Keďže pomocné polia nemáme, nevieme to spraviť takto priamočiaro. Vieme však použiť techniku z riešenia podúlohy C. Riešenie podúlohy A upravíme na funkciu, ktorá namiesto vyplnenia celého výstupného poľa len vráti jeho jeden konkrétny prvok. Takáto funkcia môže vyzerať napr. nasledovne:

Listing programu (Pascal)

```
var n : integer;                          { vstup: dĺžka postupnosti }
    A : array [1..n] of integer;          { vstup: postupnosť čísel }
    B : array [1..n] of integer;          { výstup: usporiadaná postupnosť }
    i, j, vlavo : integer;

function q_ty_najmensi_prvok_A ( q : integer ) : integer;
begin
    for i := 1 to n do begin
        vlavo := 0;
        for j := 1 to n do begin
            if (A[j] < A[i]) then inc(vlavo);
            if (A[j] = A[i]) and (j<i) then inc(vlavo);
        end;
        { tu bolo povodne toto: "B[ vlavo+1 ] := A[i];" }
        { namiesto toho tu vacsinu zapisov odignorujeme, len jeden pouzijeme }
        if (vlavo+1 = q) then q_ty_najmensi_prvok_A := A[i];
    end;
end.
```

Analogickú funkciu si môžeme napísať pre usporiadanie poľa B. No a na overenie, či pole B vzniklo preusporiadaním poľa A, nám teraz stačí pre každé i od 1 po n overiť, či sa i-ty najmenší prvok poľa A rovná i-temu najmenšiemu prvku poľa B.

Za domácu úlohu si rozmyslite, že takto naozaj dostaneme log-space program, a odhadnite jeho pamäťovú aj časovú zložitosť.

Riešenia domáceho kola kategórie B

B-I-1 Veže

Pre každú vežu vieme ľahko nájsť všetky s ktorými sa potencionálne ohrozuje. Takéto budú v tom istom riadku alebo stĺpci na šachovnici a budú opačnej farby. Nájdeme ich prejdením zoznamu veží. Každú takúto kolegyňu s ktorou sa potencionálne ohrozuje vieme skontrolovať. Ak medzi nimi leží akákoľvek ďalšia veža, nevidia na seba, inak sa naozaj ohrozujú.

Takto vyskúšame $\binom{v}{2}$ dvojíc veží. Pre každú dvojicu, ktorá sa „vidí“ (leží v tom istom riadku/stĺpci) a má opačné farby vieme v lineárnom čase prejsť všetky zvyšné veže a skontrolovať, či nestoja v ceste. Takto dostávame jednoduché riešenie s časovou zložitosťou $O(v^3)$. Celé riešenie pozostáva len z niekoľkých vnorených cyklov. Pämaťová zložitosť je $O(v)$ – stačí nám pamatať si jedno pole so súradnicami veží.

Pre jednoduchý prístup k popisu veží si údaje navonok zjednotíme v podobe štruktúry – v Pascale `record`, v C/C++ `struct`.

Podobné riešenie bez problémov vyrieši prvé tri vstupy, na zvyšné treba použiť efektívnejší prístup (alebo veľmi dlho čakať).

Listing programu (Pascal)

```
type veza = record
  riadok, stlpec, farba : longint;
end;

var n, v, i, j, k, vysledok : longint;
    ok : boolean;
    veze : array[1..100047] of veza;

function min(a, b : longint) : longint;
begin if a < b then min := a else min := b; end;

function max(a, b : longint) : longint;
begin if a > b then max := a else max := b; end;

function vidia_sa(var a, b : veza) : boolean;
begin vidia_sa := (a.riadok = b.riadok) or (a.stlpec = b.stlpec); end;

function medzi(var a, b, c : veza) : boolean;
begin
  medzi := false;
  if (a.riadok = b.riadok) and (b.riadok = c.riadok) then
    if (min(a.stlpec, b.stlpec) < c.stlpec)
```

```
      and (c.stlpec < max(a.stlpec, b.stlpec)) then medzi := true;
  if (a.stlpec = b.stlpec) and (b.stlpec = c.stlpec) then
    if (min(a.riadok, b.riadok) < c.riadok)
      and (c.riadok < max(a.riadok, b.riadok)) then medzi := true;
end;

begin
  readln(n, v);
  for i:=1 to v do
    readln(veze[i].stlpec, veze[i].riadok, veze[i].farba);
  vysledok := 0;
  for i:=1 to v do
    for j:=i+1 to v do
    if (veze[i].farba <> veze[j].farba) and (vidia_sa(veze[i], veze[j])) then
    begin
      ok := true;
      for k:=1 to v do if (medzi(veze[i], veze[j], veze[k])) then begin
        ok := false;
        break;
      end;
      if ok then inc(vysledok);
    end;
  writeln(vysledok);
end.
```

Jednoduché zlepšenie prináša pozorovanie že veža môže ohrozovať najviac štyri ďalšie: najbližšíu vľavo, vpravo, hore a dole. Pre každú vežu stačí medzi ostatnými v lineárnom čase nájsť týchto štyroch „susedov“ a skontrolovať, či sú to nepriatelia. (Počas spracúvania konkrétnej veže si pre každý smer pamätáme doteraz najbližšiu vežu od nej v tom smere.) Časovú zložitosť sme týmto zlepšili na $O(v^2)$, ale 4. a 5. vstup stále podobným prístupom rýchlo vyriešiť nevieme.

Môžeme však šikovne využiť to, že v prvých štyroch vstupoch je šachovnica malá a celá sa nám zmestí do pamäte ako dvojrozmerné pole. Môžeme si takéto pole v pamäti spraviť a zaznačiť doň všetky veže. Potom keď pre konkrétnu vežu chceme nájsť tie, ktoré ohrozuje, nemusíme prechádzať celý zoznam veží – stačí v našom poli prejsť riadok a stĺpec, v ktorých naša veža leží. Netreba zabúdať že každú dvojicu takto zarátam dvakrát – raz pre jednu vežu z dvojice, druhýkrát pre druhú, preto treba na konci vydeliť výsledný počet dvoma.

Toto riešenie má časovú zložitosť $O(nv)$. Jeho pamäťové nároky sú $O(n^2)$. Pre prvé štyri vstupy sa nám všetko do pamäte zmestí a takéto riešenie stačí na ich vyriešenie.

Ešte lepšie riešenie: Nemusíme pre každú vežu zas a znova prechádzať celý stĺpec a celý riadok. Keď už sme si veže vyplnili do dvojrozmerného poľa, stačí raz prejsť každý celý riadok a raz každý stĺpec a pre každý z nich spočítať všetky po sebe idúce dvojice veží opačnej farby. Takéto riešenie má časovú zložitosť $O(n^2)$. Nižšie uvádzame jeho implementáciu.

(Rovnakú časovú zložitosť by sme dostali, keby sme v predchádzajúcom

riešení pre každú vežu prezerali príslušný riadok/stĺpec len dovtedy, kým nenarazíme na inú vežu alebo kraj šachovnice. To preto, že každým políčkom šachovnice by sme v každom smere išli najviac jedenkrát.)

Listing programu (Pascal)

```
var n, v, i, j, riadok, stlpec, farba, posledna, h, vysledok : longint;
    sachovnica : array[1..1047, 1..1047] of longint;

begin
  for i := 1 to 1047 do
    for j := 1 to 1047 do
      sachovnica[i][j] := -1;
  readln(n, v);
  for i:=1 to v do begin
    readln(riadok, stlpec, farba);
    sachovnica[riadok][stlpec] := farba;
  end;
  vysledok := 0;
  for i := 1 to n do begin
    posledna := -1; h := -1;
    for j := 1 to n do
      if sachovnica[i][j] <> -1 then begin
        if (posledna <> -1) and (h <> sachovnica[i][j]) then inc(vysledok);
        posledna := j;
        h := sachovnica[i][j];
      end;
  end;
  for j := 1 to n do begin
    posledna := -1; h := -1;
    for i := 1 to n do
      if sachovnica[i][j] <> -1 then begin
        if (posledna <> -1) and (h <> sachovnica[i][j]) then inc(vysledok);
        posledna := i;
        h := sachovnica[i][j];
      end;
  end;
  writeln(vysledok);
end.
```

Vzorové riešenie:

Ako však vypočítať posledný vstup? Chceli by sme spraviť to isté ako v predchádzajúcom riešení – najskôr pre každý *neprázdny* riadok spočítať, koľko sa v ňom ohrozuje dvojíc veží, potom to isté spraviť pre každý *neprázdny* stĺpec.

Aby som mohol prezerať veže v poradí v akom idú na šachovnici v riadku alebo stĺpci podobne ako v predchádzajúcom riešení, potrebujem mať veže rozumne usporiadané. Usporiadajme teraz veže podľa čísla riadku a v prípade rovnosti podľa stĺpcu. V takto usporiadanom poli platí, že každý riadok zodpovedá nejakému súvislému úseku veží. Stačí teda prejsť po usporiadanom poli a pre každé dve po sebe nasledujúce veže skontrolovať, či sú v tom istom riadku a zároveň opačných farieb. Následne spravíme to isté pre stĺpce – usporiadame veže podľa stĺpcu a v prípade rovnosti podľa riadku, a potom usporiadané pole

prejdeme a spočítame ohrozujúce sa dvojice.

Celkovo takéto riešenie spotrebuje $O(v)$ pamäte a časovú zložitosť bude mať $O(v \log v)$ kvôli triedeniu. V moderných programovacích jazykoch môžeme na triedenie použiť knižničnú funkciu:

Listing programu (Python)

```
def rataj_riadky(veze):
    veze.sort()
    answer = 0
    for i in range(len(veze)-1):
        if veze[i+1][0]==veze[i][0] and veze[i+1][2]!=veze[i][2]:
            answer += 1
    return answer

from sys import stdin
N = int(stdin.readline())
V = int(stdin.readline())

veze = [ tuple( int(x) for x in stdin.readline().split() ) for v in range(V) ]
total = rataj_riadky(veze)

veze = [ (s,r,c) for r,s,c in veze ]
total += rataj_riadky(veze)

print(total)
```

Na utriedenie 100 000 prvkov potrebujeme dostatočne rýchly triediaci algoritmus, ideálne bežiaci v čase rádovo $n \log n$ pre n prvkov. Veľa programovacích jazykov má implementované takéto algoritmy, napríklad `sort` v C++. V ostatných jazykoch je treba naprogramovať vlastné rýchle triedenie. My si stručne popíšeme jeden zo zaužívaných prístupov – HeapSort, teda triedenie pomocou haldy.

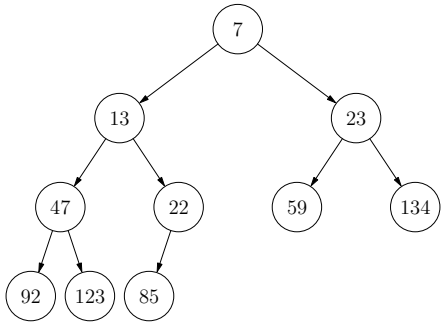
Halda:

Predstavme si, že máme čiernu krabičku s tlačidlom, do ktorej vieme vhadzovať čísla a z ktorej po každom stlačení tlačidla vypadne najmenšie z čísel, ktoré sú práve v nej.

Takáto krabička vie byť celkom užitočná. Pomocou nej by sa nám napríklad ľahko triedilo – nahádžeme všetky čísla do nej, no a potom už len stláčame tlačidlo, kým postupne nevypadnú všetky v utriedenom poradí.

Jednou šikovnou implementáciou takejto krabičky je halda. Je to vlastne binárny strom, ktorý má každé poschodie úplne plné, možno okrem toho posledného, najspodnejšieho. Každý prvok, okrem tých najspodnejších, má teda práve dvoch synov. Prvky musia byť usporiadané tak, aby platilo, že hodnota uložená v ľubovoľnom vrchole je menšia alebo rovná ako každá z hodnôt uložená v jeho synoch (ak nejakých má).

Takto môže vyzerať halda:



Všimnime si zaujímavú vlastnosť haldy: najmenšie číslo je určite v jej koreni. O ostatných číslach už toho tak veľa nevieme povedať – všimni si, že napríklad 22 je hlbšie ako 23.

Na to, že halda je strom, môžeme zase šťastne zabudnúť. Haldu si totiž vieme úplne jednoducho pamätať v poli. Stačí si vrcholy očíslovať po vrstvách. Na políčku s číslom x teda bude prvok, ktorý by sme prečítali ako x-tý, keby sme haldu „čítali po riadkoch“.

Všimnime si niektoré vlastnosti takto uloženej haldy:

- Ak je v halde n prvkov, v poli budú na políčkach 1 až n.
- Koreň haldy, teda najmenší prvok v nej, je na políčku s číslom 1.
- K-ta vrstva haldy sa v poli začína na políčku s indexom 2^{K-1}.
- Synovia vrcholu s číslom x majú vždy čísla $2x$ a $2x+1$.
- A opačne, otec vrcholu s číslom x má číslo $\lfloor x/2 \rfloor$.

Halda z obrázku by vyzerala v poli takto:

1	2	3	4	5	6	7	8	9	10	11	12	
7	13	23	47	22	59	134	92	123	85			

Ukážeme teraz, ako do haldy efektívne pridávať nové čísla a ako z nej efektívne vyberať najmenšie.

Vloženie prvku:

Ako teda vložíme nejaký prvok do haldy? Zaradíme ho do poľa na prvé voľné miesto. Jediné, čo nám teraz môže kaziť „haldovitosť“, je, že tento prvok môže byť menší od prvku nad ním. V tom prípade túto dvojicu prvkov vymeníme. Rozmyslite si, že opäť môže nastať jediný problém: nový prvok môže byť menší

aj od svojho nového otca. V takomto prípade postup opakujeme. (Tomuto sa hovorí „bublanie prvku dohora“.)

Tento postup určite skončí, prinajhoršom vtedy, ked sa nový prvok dostane na úplný vrch haldy – do koreňa. Po jeho skončení je opäť celá halda v poriadku, úspešne sme teda vložili nový prvok.

Výber najmenšieho prvku:

A čo s vybratím najmenšieho prvku? To bude fungovať podobne. Vieme, že najmenší prvok je ten na vrchu haldy. Tentokrát ho ale nemôžeme len tak odstrániť, vznikla by nám tam totiž diera. No a tú treba niečím zaplniť. Najjednoduchšie riešenie: Zoberieme posledný prvok v poli (t. j. najpravejší list v poslednej vrstve) a presunieme ten na začiatok poľa.

Touto zmenou sme opäť dosiahli, že čísla máme uložené na prvých niekoľkých políčkach poľa. Nemusí to ale ešte byť korektná halda. To, čo nám ju môže kaziť, je práve presunutý prvok. Preto zopakujeme niečo podobné, ako pri vkladaní. Tentokrát ale presunutý prvok môže byť len priveľký, preto ho budeme musieť „prebublať“ dodola. Toto bublanie treba robiť trochu šikovnejšie. Tentokrát totiž náš „zlý“ prvok môže mať až dvoch synov a byť väčší od každého z nich. Hravo ale zistíme, že riešenie je jednoduché: stačí ho vymeniť s menším z oboch synov.

Opäť, tento postup je konečný. Skončíme, ak už sú obidvaja aktuálni synovia väčší alebo rovní presunutému prvku, prípadne ak sa náš prvok prebublal až do najspodnejšej vrstvy. V každom prípade máme opäť korektnú haldu.

Odhad zložitosti:

Všimnime si, čo sa deje pri jednom prebublaní prvku. Pri vkladaní prebubleme v najhoršom z poslednej vrstvy až do koreňa, pri vyberaní minima naopak, z koreňa až po najhlbšiu vrstvu. V obidvoch prípadoch je počet operácií úmerný hĺbke stromu, teda počtu vrstiev. A aká je tá hĺbka? Na zaplnenie k vrstiev haldy potrebujeme $2^k - 1$ prvkov, preto halda s n prvkami má približne $\log_2 n$ vrstiev. Každá operácia s haldou, v ktorej je n prvkov, má teda časovú zložitosť $O(\log n)$.

Triedenie pomocou haldy:

Ako sme už spomínali na začiatku, pomocou haldy vieme ľahko napísať triedenie, známe pod menom HeapSort. (Heap je halda po anglicky.) Pri triedení n prvkov najskôr n-krát vložíme prvok do haldy, potom odtiaľ n-krát vyberieme najmenší. Počas celého tohto procesu nie je nikdy v halde viac ako n prvkov, preto časová zložitosť každej operácie s ňou je $O(\log n)$. Týchto operácií je $2n$,

preto je výsledná časová zložitosť HeapSortu $O(n \log n)$. Pamäťová zložitosť haldy aj triedenia pomocou nej je samozrejme $O(n)$.

B-I-2 Dosah

Naša úloha sa skladá z viacerých otázok, ale sú od seba dosť nezávislé, preto každú takúto otázku budeme riešiť samostatne. Nech teda máme už danú mapu a konkrétne x, y a k. Chceme zistiť, kam všade sa vie tank dostať tak, aby neprešiel viac ako k krokov a zároveň neprešiel cez prekážku alebo nevyšiel z mapy.

Namiesto toho, aby sme z tohto políčka púšťali tančík a určovali mu všetky možné cesty, vylejme na tento štvorec vedro vody a sledujme čo sa deje. Na začiatku je voda, iba na políčku (x, y), postupne sa však začne vylievať aj na susedné políčka. Presnejšie bude sa snažiť vyliať na horné, dolné, pravé a ľavé políčko. Po prvom kroku je teda voda na pôvodných súradniciach (x, y) a tiež na nových súradniciach $(x+1, y)$, $(x-1, y)$, $(x, y+1)$ a $(x, y-1)$ – samozrejme ak na týchto miestach nie je prekážka alebo nebodaj nie sú mimo mapy. Toto vylievanie samozrejme pokračuje, tentoraz sa voda šíri ďalej zo všetkých už zaliatych políčok.

Teraz si všimnime, že pre každé n platí, že po n vyliatiach na susedné políčka je voda presne na tých miestach, kam sa vie dostať náš tank z (x, y) na najviac n posunutí. A keby sme nasimulovali toto rozlievanie vody, vieme potom ľahko zrátať odpoveď.

Pomalšie riešenie: Prvé, čo vieme spraviť, je naozaj priamočiare simulovanie vyššie popísaného procesu. Zoberieme si dvojrozmerné pole veľkosti $r \times s$ a do každého políčka si zaznačíme hodnotu -1, ktoré predstavuje, že na toto políčko sa voda ešte nedostala. Iba na políčko (x, y) si uložíme číslo 0, lebo tu je na začiatku voda. Ďalej postupne pre každé n od 1 po k zopakujem nasledovný postup:

Prejdem celé dvojrozmerné pole. Ak na nejakom políčku nájdem číslo od 0 po $n-1$, znamená to, že už je na tomto políčku voda, ktorá sa chce rozlievať ďalej. Všetkým jeho voľným susedom (teda nie je to prekážka, je na mape a ešte tam nebola voda – v poli mám uložené -1) zaznačím do poľa hodnotu n. To znamená, že toto políčko bolo zaliate v n-tom kroku.

Tento postup nám zaručí, že po n-tej iterácii máme v poli nezáporné čísla na tých miestach, kam sa vie tank dostať na najviac n krokov. Po kroku, v ktorom

$n = k$, teda máme označené práve tie políčka, ktoré nás zaujímajú. Prejdeme celú mapu, spočítame, koľko ich je a vypíšeme toto číslo na výstup.

Bohužiaľ musíme počas simulácie rozlievania vody k-krát prejsť cez celú mapu, a teda máme časovú zložitosť $O(krs)$ na zodpovedanie jednej otázky.

Vzorové riešenie: Chceli by sme to teda nejak zrýchliť. Pozrime sa, či v predchádzajúcom programe nerobíme niečo zbytočne. Naozaj nám treba pozerať na všetky políčka? Ľahko nahliadneme, že nie. Ak sme v n-tej iterácii, nepotrebujem kontrolovať rozlievanie vody zo všetkých políčok, ale len z tých, kam sa voda prvýkrát dostala v predchádzajúcom, $(n-1)$. kroku. Všetko, čo sa dalo zaliať skôr, už dávno aj zaliate je. Inými slovami, nikdy sa nemôže stať, že sa čísla v dvoch susedných voľných políčkach budú líšiť o viac ako 1 – akonáhle niekedy zalejem jedno z nich, najneskôr v nasledujúcom kole (z neho) zalejem aj to susedné. V každej iterácii rozlievania nám teda stačí kontrolovať políčka s číslom $n-1$.

Otázkou však je, ako to robiť dostatočne rýchlo. Efektívne riešenie tejto úlohy je známe pod názvom prehľadávanie do šírky.

Prehľadávanie do šírky:

Algoritmus je v podstate simuluje šírenie vody, akurát to robí veľmi šikovne. Podstatou je dátová štruktúra fronta, do ktorej vieme prvok vložiť a potom ho aj vybrať, pričom vo vnútri v dátovej štruktúre to funguje ako klasický rad na obed. Keď chceme vložiť nový prvok, zaradíme ho na koniec tohto radu, a keď prvok vyberáme, vyberieme vždy prvý prvok – ten, ktorý už najdlhšie čakal.

V našej fronte budú na spracovanie čakať políčka, z ktorých chceme rozlievať vodu. Na začiatku je tam teda jediné políčko: (x, y).

Kým je vo fronte nejaké políčko, robíme nasledujúcu vec: Vyberieme z fronty políčko, z ktorého sa ide šíriť voda a do premennej n si poznačím, v ktorom kroku som toto políčko zaplavil (to mám poznačené v poli). Pozrieme sa na susedné políčka. Ak je niektoré z nich voľné, zaznačíme si, že sme toto políčko objavili v kroku $n+1$ a vložíme ho do fronty na spracovanie. (Samozrejme, ak $n = k$, už nové políčka na spracovanie do fronty nevkladáme.)

Po dobehnutí tohto algoritmu dostávame presne rovnaký výsledok ako pred tým, teda si zrátame počet zaplavených miest a vrátime ho ako výsledok.

A prečo to funguje? Všimnime si, že na začiatku sú vo fronte miesta, ktoré sa dajú zaplaviť na 0 krokov – také je jedno. Potom do fronty vložím všetky, ktoré sa dajú zaplaviť na 1 krok. Tieto postupne z fronty vyberám a do fronty *za ne* pridávam políčka, ktoré sú zo štartu dosiahnuteľné na 2 kroky. Keď teda

spracujem všetky políčka dosiahnuteľné na 1 krok, vo fronte budú na spracovanie čakať práve všetky políčka dosiahnuteľné na 2 kroky. A tak ďalej. Teda robím to isté čo predtým, nepozerám sa však na všetky políčka znovu a znovu, ale každé políčko spracujem najviac jedenkrát – vtedy, keď to treba.

Zostáva už len určiť pamäťovú a časovú zložitosť. Pamäť je $O(rs)$, lebo si musím pamätať ako vyzerá celá mapa. Čas potrebný na odpovedanie na jednu otázku je $O(rs)$, lebo každé miesto zaplavím najviac raz a teda sa aj najviac raz ocitne vo fronte.

Listing programu (Python)

```
from sys import stdin
from queue import Queue
T = int(stdin.readline())

for t in range(T):
    R, S, Q = [ int(x) for x in stdin.readline().split() ]
    A = [ [ int(x) for x in stdin.readline().split() ] for r in range(R) ]

    for q in range(Q):
        r0, s0, krokov = [ int(x) for x in stdin.readline().split() ]
        dist = [ [ 1234567890 for s in range(S) ] for r in range(R) ]
        dist[r0-1][s0-1] = 0
        Q = Queue()
        Q.put( (r0-1,s0-1) )
        answer = 1

        while not Q.empty():
            cr,cs = Q.get()
            for nr,ns in [ (cr+1,cs),(cr-1,cs),(cr,cs+1),(cr,cs-1) ]:
                if nr<0 or ns<0 or nr>=R or ns>=S: continue
                if dist[nr][ns] <= dist[cr][cs]+1: continue
                if A[nr][ns] == 1: continue
                if dist[cr][cs] == krokov: continue
                answer += 1
                dist[nr][ns] = dist[cr][cs]+1
                Q.put( (nr,ns) )
        print(answer)
```

B-I-3 Súčet súčtov

Čo by ste robili vy, keby vám váš učiteľ matematiky dal za úlohu sčítať postupnosti čísel za celú triedu? Určite by sa mnohým z vás nechcelo robiť toľko sčítaní, a nejako by ste sa snažili ušetriť si robotu. Ale ako?

Najskôr sa pozrime, koľko výpočtov by žiaci spravili, keby počítali všetko poctivo tak, ako bolo uvedené v zadaní. Máme rádovo n^2 žiakov, z ktorých každý spraví rádovo n sčítaní, čiže dokopy máme zložitosť $\Theta(n^3)$. (Dá sa spočítať, že presný počet sčítaní je $n(n+1)(n+2)/6 - 1$.)

Ako teda tento postup zefektívniť? Napríklad nemá veľký zmysel počítať niečo viackrát. Keď máte za úlohu sčítať prvých sedemnásť čísel a zistíte, že váš spolužiak už má vypočítaný súčet prvých šestnástich, tak nemusíte jeho robotu opakovať, ale sa ho môžete spýtať na jeho výsledok a pripočítať už len posledné, sedemnáste číslo.

Celý postup teda môžeme zlepšiť napríklad nasledovne: Žiaci rozdelia do skupín podľa toho, kde začína ich úsek. (Teda v prvej skupine budú tí, ktorí majú spočítať čísla od prvého po nejaké, v druhej skupine tí, ktorí začínajú sčitovať od druhého čísla, a tak ďalej.) V rámci skupiny sa potom postavia do radu podľa toho, ktorým číslom končia.

Počítanie bude teraz fungovať nasledovne: Prvý žiak v rade má sčítať len jednoprvkovú postupnosť. Teda rovno vie výsledok. Výsledok povie ďalšiemu žiakovi. Ten následne k nemu pripočíta nasledujúce číslo z postupnosti a tiež je hotový. Výsledok povie nasledujúcemu v rade, a tak ďalej. Každý žiak takto spraví len jedno jediné sčítanie – pripočíta svoje číslo ku súčtu žiaka pred ním. (Ak napríklad mám počítať súčet od 3. čísla po 11., počkám si, kým mi predo mnou stojaci žiak oznámi súčet od 3. po 10. číslo, pripočítam k nemu 11. číslo a som hotový.)

Takýmto spôsobom dokáže každý žiak zistiť svoj súčet v konštantnom čase. Keďže žiakov je rádovo n^2, spraví sa pritom len $O(n^2)$ operácií. Pri implementácii nám stačía dva vnorené cykly: vonkajší určuje začiatok úseku, vnútorný zase koniec (teda žiaka, ktorý práve počíta).

Listing programu (C++)

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

int main(){
    int n;
    cin >> n;
    long long celkovysucet = 0;

    //nacitame postupnost
    long long postupnost[n+1];
    for(int i = 1; i <= n; ++i) cin >> postupnost[i];

    for(int i = 1; i <= n; ++i) {
        // spocitame ziakov, ktorych postupnosti zacinaju i-tym prvkom
        long long aktualny = 0;
        for(int j = i; j <= n; ++j) {
            // nova hodnota = hodnota ziaka predo mnou + j-ty clen postupnosti
            aktualny += postupnost[j];
            celkovysucet += aktualny;
        }
    }
    cout << celkovysucet << endl;
}
```

Ako sme videli, keď sa snažíme nejaký postup zrýchliť, tak sa zamyslíme, či nerobíme niečo zbytočne veľakrát. V predchádzajúcom riešení už toho zdanlivo nejde veľa zlepšiť – pre každého konkrétneho žiaka sme predsa jeho výsledok zistili v konštantnom čase a lepšie to už nemá ako ísť, nie?

Lenže ono to efektívnejšie pôjde. Trik bude v tom, že my vlastne jednotlivé medzivýsledky nepotrebujeme. Nás zaujíma len ich súčet. A nič nebráni tomu, aby sme tento súčet vedeli nejakým iným, šikovnejším spôsobom vypočítať efektívnejšie.

Predstavte si, že chcete sčítať 100-krát číslo 47. Čo by ste naozaj urobili? Urobili by ste 99 operácií sčítania alebo niečo iné? Asi nik by nesčítal, všetci predsa vedia, že výsledok bude 100 krát 47, čiže 4700.

Podobne aj v tejto úlohe budeme vo vzorovom riešení namiesto opakovaného sčítania radšej násobiť. Pre každý prvok postupnosti si jednoducho spočítame, koľkokrát sa objaví vo výsledku. Inými slovami, to, čo potrebujeme pre každý prvok zistiť, je počet žiakov, ktorí tento prvok majú vo svojom úseku.

Keď si zoberieme nejaké dve čísla $a, b; 1 \leq a \leq b \leq n$, tak existuje práve jeden žiak, ktorý mal za úlohu sčítať čísla z úseku od a po b vrátane. Zoberme si nejaký konkrétny, povedzme i-ty prvok postupnosti. Úsek od a po b, ktorý tento prvok obsahuje, musel zjavne začať najneskôr na i-tom mieste, teda musí platiť $a \leq i$. Zároveň tento úsek musí skončiť až po i-tom prvku, teda pre b platí $i \leq b$.

Úsek, ktorý obsahuje i-ty prvok, má teda i možných začiatkov a $n - i + 1$ možných koncov. Každý začiatok s každým koncom určuje jeden konkrétny úsek. Preto je takýchto úsekov presne $i \times (n - i + 1)$. Inými slovami, i-ty prvok sa vyskytne v celkovom súčte práve $i(n - i + 1)$-krát.

Program, ktorý rieši túto úlohu, teda postupne prečíta celý vstup od prvku s číslom 1 až po n, pre každé i vynásobí hodnotu i-teho prvku počtom $i \times (n-i+1)$, a všetky tieto medzivýsledky sčíta.

Celková časová zložitosť bude $O(n)$. Tú už zjavne nevieme zlepšiť, lebo musíme načítať celý vstup. Pamäťová zložitosť bude $O(1)$, keď si uvedomíme, že pole si vlastne ani netreba pamätať. Jediné, čo potrebujeme, je aktuálne spracúvané číslo, jeho index, číslo n a doterajší súčet.

Listing programu (C++)

```
#include <iostream>

int main(){
    long long n;
    std::cin >> n;
```

```
    long long sucet = 0;
    for (int i = 1; i <= n; ++i) {
        long long prvok;
        std::cin >> prvok;
        sucet += prvok * i * (n-i+1);
    }
    std::cout << sucet << "\n";
}
```

B-I-4 Paralelné trampoty

V riešení tejto úlohy si ukážeme, že vo svete, v ktorom sa naraz deje viacero vecí, môže často vzniknúť nečakaný chaos.

Podúloha A – 6 bodov:

Vyrobiť na tabuli číslo 900 je ľahké. Stačí napríklad, aby postupne každé dieťa 30× zopakovalo postup zo zadania. Presnejšie, akýkoľvek scenár, pri ktorom sa jednotlivé vykonania postupu zo zadania neprekrývajú, vyrobí na tabuli číslo 900. Ak ho chceme zmenšiť, musíme teda nechať viac detí vykonávať ich postup v tom istom čase. Tu je prvé možné zlepšenie, ktoré nás dostane z 900 na 30:

- Postupne po jednom vôjde každé dieťa do triedy, prečíta z tabule 0 a zapamätá si ju.
- Všetky deti stoja pred triedou a usilovne počítajú, až kým každé z nich nedostane výsledok 1.
- Postupne po jednom vôjde každé dieťa do triedy, zotrie tabuľu a napíše na ňu svoj výsledok: číslo 1.
  V tomto okamihu už máme za sebou 30 vykonaní postupu zo zadania – no na tabuli namiesto čísla 30 svieti len číslo 1.
- Ešte 29× zopakujeme predchádzajúce tri body.

Celkom slušné zlepšenie, nie? Aj keby sme namiesto 30 detí mali v našej triede 1000, aj tak by sme skončili na čísle 30 (a nie 30000).

No ale lepšie to už zjavne nepôjde. Všimnime si totiž konkrétne dieťa. To svoj postup vykoná 30-krát, a to nutne postupne. Zakaždým pritom zvýši hodnotu na tabuli. A teda na konci musí byť hodnota na tabuli aspoň 30. A tým sme skončili – ukázali sme, že 30 dosiahnuť vieme, aj to, že menej nie.

Presvedčil vás predchádzajúci odsek? Ak tušíte zradu, tušenie vás veru neklame. V „dôkaze“, že menej ako 30 nevieme dosiahnuť, je totiž chyba. Kde? Pozrime sa na dotyčné jedno dieťa poriadnejšie. A vyberme si napríklad sedemnáste opakovanie jeho postupu. Naše dieťa vošlo do triedy, zapamätalo si číslo x z tabule, vyšlo von, prirátalo k nemu 1, znova vošlo dnu, zotrelo tabuľu, napísalo $x+1$ a zase vyšlo von. A teraz prichádza kameň úrazu. Totiž skôr, ako toto dieťa začne svoje osemnáste kolo, môže sa pokojne stať, že do triedy vbehne iné dieťa so svojim výsledkom – a pokojne sa môže stať, že tento výsledok je menší ako $x+1$.

Teda síce *je pravda*, že konkrétne dieťa postupne 30-krát zvýši aktuálnu hodnotu na tabuli, ale už *nie je pravda*, že ide o tú istú, postupne zvyšovanú hodnotu.

Ak to ešte nie je jasné, ukážeme to najskôr na malom príklade:

- Adamko bol v triede a zapamätal si 0.
- Betka bola v triede a zapamätala si 0.
- Cilka bola v triede a zapamätala si 0.
- Adamko bol v triede a zapísal 1.
- Dušanko bol v triede a zapamätal si 1.
- Dušanko bol v triede a zapísal 2.
- Betka bola v triede a zapísala 1.
- Dušanko bol v triede a zapamätal si 1.
- Dušanko bol v triede a zapísal 2.
- Cilka bola v triede a zapísala 1.
- Dušanko je už tretíkrát v triede po číslo...
  ... a po tretíkrát vidí na tabuli číslo 1.
- Dušanko tretíkrát zapísal na tabuľu číslo 2.

No a teraz už je čas na optimálne riešenie. Jeho hodnota je prekvapivá: najmenšie možné číslo, ktoré môže na konci ostať na tabuli, je číslo 2. Existuje viacero spôsobov, my si ukážeme jeden z nich. Dve z detí si nazveme Viktor a Zuzka, ostatných 28 ponecháme bez mena.

- Viktor bol v triede a zapamätal si 0.
- Zuzka bola v triede a zapamätala si 0.
- Všetky ostatné deti 30× vykonali celý postup (v ľubovoľnom poradí).
- Viktor bol v triede a zapísal 1.
- Viktor vykonal svoje 2. až 29. opakovanie. Na tabuli je číslo 29.

- Zuzka bola v triede a zapísala 1.
- Viktor bol v triede a zapamätal si 1.
- Zuzka vykonala svoje 2. až 30. opakovanie. Na tabuli je teraz číslo 30.
- No a úplne na záver prišiel do triedy Viktor a zapísal na tabuľu číslo 2.

Do kompletného riešenia už chýba len dôkaz, že menej ako 2 sa už naozaj dosiahnuť nedá. Ten je našťastie ľahký: Nula je na tabuli len na začiatku. Od okamihu, kedy prvýkrát niekto zapíše svoj výsledok, je číslo na tabuli stále kladné. Všimnime si dieťa, ktoré na konci zapísalo úplne poslednú hodnotu na tabuľu. Ide o jeho tridsiate opakovanie postupu, a teda číslo, ktoré v ňom prečítalo z tabule, bolo nutne už kladné – a teda aspoň 1. Tým pádom zapísaný výsledok je aspoň 2, č.b.t.d.

Podúloha B – 4 body:

Pre pripomenutie, tu je ešte raz zadanie súťažnej úlohy. V Marienkinom domovskom adresári je súbor `/home/marienka/ponik3.png`, ku ktorému Janko nemá prístup. Správca Samko však spravil program, ktorý funguje nasledovne: keď ho užívateľ X spustí ako „`posli /nejaka/cesta/subor email`“, program postupne:

1. otvorí adresár `/nejaka/cesta` a skontroluje, či X má právo čítať `subor`.
2. ak áno, otvorí `subor`, načíta ho a prepošle ho na adresu `email`.

Trik je samozrejme vo vhodnom použití symbolických liniek, vysvetlených v zadaní. Asi najjednoduchšie riešenie vyzerá nasledovne:

1. Janko vyrobí symbolickú linku ukazujúcu na súbor v jeho domovskom adresári.
2. Janko spustí Samkov program, pričom ako prvý parameter použije práve dotyčnú symbolickú linku.
3. Samkov program skontroluje, že Janko má právo čítať dotyčný súbor. Všetko vyzerá OK.
4. Janko rýchlo zmení svoju symbolickú linku tak, aby ukazovala na Marienkin obrázok poníka.
5. Samkov program ho prečíta a pošle Jankovi mailom.

Napriek tomu, že ide o nesmierne jednoduchú formu útoku, sú takéto útoky na bezpečnosť systémov v praxi pomerne bežné. Mnohé *exploity* (programy zneužívajúce slabiny systémov) sú založené práve na tom, že útočník odhalí

v systéme takúto chybu. V zadaní sme už spomínali, že sa takýmto chybám, vznikajúcim počas súčasného behu viacerých programov, zvykne hovoriť *race condition*. Slovo *race* (závod) je v názve práve preto, že sa programy pretekajú – stihne útočník v správnej chvíli urobiť potrebné zmeny?

Vo výhode je samozrejme útočník. Totiž často má k dispozícii pokusov, koľko len chce, pričom mu stačí uspieť jeden jediný raz. A skutočne aj mnohé reálne exploity z praxe fungujú jednoducho tak, že sa dokola skúšajú „trafiť do správneho okamihu“, až sa im to skôr či neskôr aj podarí.

Riešenia krajského kola kategórie A

A-II-1 Výkrm

Z celého kráľovstva sa začali zbiehať odvážni junáci a junáčky, ktorí chceli kráľovskému páru pomôcť. Prvý odvážlivec predstavil kráľovnej Totňi nasledovné riešenie: Každý návrh skontrolujeme osobitne. Pre každé jedlo v ňom skúsime vybrať vhodné mesto, v ktorom ho má Hurim zjesť. Kde by mal zjesť prvé jedlo z plánu? Ak má na výber viacero miest, kde ho podávajú, zjavne *nič nepokazíme*, ak mu ho dáme v meste s najmenším číslom – ostane nám tak pri plnení zvyšku plánu na výber najviac iných miest. A v tejto úvahe môžeme pokračovať aj ďalej. Vždy, keď chceme Hurimovi dať nejaké ďalšie jedlo, ideme z mesta, kde Hurim naposledy jedol, ďalej, až kým nestretneme *prvé* také, kde sa toto konkrétne jedlo podáva. Ak sa takto dostaneme až na koniec kráľovstva a práve hľadané jedlo už nikde nepodávali, návrh vyhlásime za nezrealizovateľný.

Je zjavné, že ak sa nám podarí nájsť ku každému jedlu mesto, kde ho Hurim zje, môžeme oprávnene prehlásiť návrh za zrealizovateľný. Kráľovná Totňa však mala ešte isté pochybnosti, či sa niekedy nemôže stať, že by sme dostali dobrý rozvrh, ale prehlásili ho za nerealizovateľný. A tak odvážlivec pod hrozbou sťatia hlavy musel dokázať kráľovnej, že pre každý zrealizovateľný plán náš algoritmus naozaj jednu možnú realizáciu nájde.

Myšlienku dôkazu sme si naznačili už vyššie. Sporom. Nech teda existujú nejaké spôsoby realizácie, ale náš algoritmus žiaden nenájde. Zo všetkých realizácií si vyberieme „najmenšiu“, teda tú, ktorá má na začiatku čo najviac krokov spoločných s čiastočným plánom, ktorý našiel náš algoritmus. V nejakom kroku sa potom ale líšia – náš algoritmus chcel vybrať nejaké mesto m_1, zatiaľ čo zvolená realizácia použije iné mesto m_2. Zjavne musí platiť $m_1 < m_2$, lebo m_1 je prvé mesto, ktoré sa v danej situácii dalo použiť. To ale znamená, že v nami zvolenej realizácii môžeme *namiesto* mesta m_2 použiť mesto m_1 – a to je spor s tým, že sme na začiatku vybrali realizáciu, ktorá sa najdlhšie zhodovala s našim algoritmom.

(Alebo iný pohľad na vyššie uvedený dôkaz: ak máme ľubovoľnú korektnú realizáciu, vieme ju prerobiť na tú, ktorú nájde náš algoritmus – a to tak, že postupne ideme po jednotlivých jedlách plánu a zakaždým skontrolujeme, či nemôžeme zjedenie daného jedla posunúť do skoršieho mesta.)

A tak náš prvý odvážlivec našiel funkčný algoritmus, pracujúci v čase $O(k \cdot (n + \ell))$: pre každý z k plánov postupne prechádza cez všetkých n miest a

odškrtáva si, ktoré z ℓ jedál daného plánu už Hurim zjedol. A keďže zjavne má zmysel uvažovať len situácie, kde $\ell \leq n$, môžeme tento odhad zjednodušiť na $O(kn)$.

Vám takýto algoritmus mohol pomôcť k piatim bodom, kráľovnej však nestačil, lebo bol príliš pomalý a kým by sa dočkala výsledku, bolo by neskoro.

Našťastie sa našiel niekto ďalší, kto prišiel na šikovné vylepšenie: Pre každé jedlo vytvoríme zoznam miest, v ktorých je toto jedlo špecialita. Potom keď kontrolujeme konkrétny plán, nemusíme ísť zaradom po celom kráľovstve. Keď vieme, aké jedlo má Hurim zjesť ako ďalšie v poradí, môžeme si v zozname jeho výskytov nájsť prvé vhodné mesto (t.j., mesto s číslom väčším ako to, kde práve sme) pomocou binárneho vyhľadávania.

Keďže miest je dokopy n, každý zoznam výskytov má dĺžku najviac n, a teda čas potrebný na nájdenie nasledujúceho výskytu práve hľadaného jedla vieme zhora odhadnúť ako $O(\log n)$.

Celkovo sa takto časová zložitosť riešenia zlepší na $O(n + k\ell \log n)$: najskôr načítame celý vstup, a potom pre každý z k plánov a každé z ℓ jedál v ňom použijeme jedno binárne vyhľadávanie na nájdenie správneho nasledujúceho mesta (ak existuje). Za takéto riešenie sa dalo získať až deväť bodov. Kráľovnej už stačilo, my sa s ním však neuspokojíme a ukážeme si, ako sa zbaviť ešte aj toho logaritmu.

Jeden možný spôsob, ako sa logaritmu zbaviť, poddaní vymysleli rýchlo, ale fungoval len pre malé m: namiesto binárneho vyhľadávania si jednoducho pre každú dvojicu (mesto,jedlo) predpočítame, v ktorom najbližšom meste s väčším číslom toto jedlo podávajú. Pre konkrétne jedlo toto predpočítanie vieme spraviť v čase $O(n)$ prechodom cez všetky mestá (od konca), dokopy má teda predpočítanie časovú aj pamäťovú zložitosť $O(mn)$. Následne už vieme každú položku každého plánu spracovať v konštantnom čase, celková časová zložitosť je teda $O(mn + k\ell)$. Za takéto riešenie sa dalo získať sedem bodov.

Nakoniec predsa len prišiel aj poddaný, ktorý dokázal nájsť optimálne riešenie. Pochopil, že pre dosiahnutie úspechu musíme kontrolovať všetky plány naraz. Aby sme to vedeli robiť efektívne, budeme si v plánoch udržiavať poriadok, a preto budú roztriedené do m šuflíkov, podľa toho, ktorým jedlom začínajú. Počas kontroly budeme z plánov odtrhávať začiatok, a následne ich preskupovať do nových šuflíkov podľa toho, čo nové sa na ich začiatku ocitlo.

Po úvodnom rozdelení do šuflíkov sa pozrieme na zoznam miest, a budeme dokola opakovať nasledovné: Nech sa v práve spracúvanom meste kráľovstva varí

jedlo x. Toto jedlo Hurim zje práve vtedy, keď je na začiatku jeho plánu. Preto zoberieme všetky plány, ktoré začínajú číslom x (máme ich v x-tom šuflíku), z každého odtrhneme začiatočnú položku, ktorú sme práve splnili, a rozhádžeme ich do správnych nových šuflíkov podľa novej začiatočnej položky.

Plány, ktoré sme celé poodtŕhali, sú realizovateľné, ostatné nie. Dôkaz vyplýva z toho, že vlastne len naraz pre všetky plány robíme jednoduchý pažravý algoritmus, ktorého správnosť sme si dokázali na začiatku tohto vzorového riešenia.

Čo sa týka samotnej implementácie, jedna možnosť je, že šuflíky, zoznam miest, a plány budú obyčajné polia. A ku každému poľu budeme mať ešte premennú, ukazujúcu na jeho aktuálny „začiatok“, teda miesto, kde začína ešte nespracovaná časť. V šuflíku si samozrejme pamätáme len *indexy* plánov, ktoré práve obsahuje. Vďaka tomu vieme robiť všetky potrebné operácie v konštantnom čase.

Oveľa pohodlnejšie a rovnako rýchle bude použiť na implementáciu šuflíkov, zoznamu miest a plánov fronty. Zjednoduší sa nám tým odtrhávanie začiatku.

V oboch prípadoch si však treba dávať pozor na jednu vec: Pokiaľ presúvame plán zo šuflíka späť do toho istého šuflíka, nechceme, aby sa spracoval znova. (To je ekvivalentné s tým, že nedovoľujeme dve zastávky v tom istom meste.)

Celková zložitosť algoritmu bude $O(n + k\ell)$ pretože každá položka bude odtrhnutá práve raz a na jej odtrhnutie vynaložím konštantný čas. Položiek je n na zozname miest a $k\ell$ v plánoch. Táto časová zložitosť je zjavne optimálna, keďže toľko času potrebujeme už len na samotné načítanie vstupu. Pamäťová zložitosť je tiež $O(n+k\ell)$, pretože si všetky plány aj zoznam miest potrebujeme naraz pamätať.

Listing programu (C++)

```
// Fruit of Light; Apple Strawberry
#include<cstdio>
#include<algorithm>
#include<vector>
#include<queue>
using namespace std;
typedef queue<int> fronta;

int n,m,k,l,a;
fronta mesta;
vector<fronta> sufliky, plany;

int main(){
    // NAČÍTAME VSTUP, VYTVORÍME FRONTY
    scanf("%d%d%d%d",&n,&m,&k,&l);
    for(int i = 0; i<n; ++i) { scanf("%d",&a); mesta.push(a); }
```

```
    plany.resize(k);
    sufliky.resize(m+1);
    for(int i = 0; i<k; ++i){
        for (int j = 0; j<l; ++j) { scanf("%d",&a); plany[i].push(a); }
        sufliky[plany[i].front()].push(i);
    }

    // SKONTROLUJEME PLÁNY
    while(!mesta.empty()){
        int x = mesta.front();
        mesta.pop();
        int pocet = sufliky[x].size();
        while (pocet--){
            int i = sufliky[x].front();
            sufliky[x].pop();
            plany[i].pop();
            if (!plany[i].empty()) sufliky[plany[i].front()].push(i);
        }
    }

    //VYPÍŠEME REALIZOVATEĽNÉ PLÁNY
    for(int i = 0; i<k; ++i) printf("%s\n",(plany[i].empty())?"ano":"nie");
}
```

A-II-2 Vyvážené jablone

V celom vzorovom riešení používame takú orientáciu jablone ako v zadaní. Koreň je teda úplne na spodku a z každého uzlu idú dve vetvy *dohora*. Podstrom určený uzlom u bude tá časť jablone, do ktorej z koreňa musíme ísť cez uzol u. Aj samotný uzol u patrí do tohto podstromu.

Pri riešení súťažnej úlohy je najjednoduchšie postupovať od listov ku koreňu. Samotný list je vždy vyvážený. Uzol, z ktorého idú dohora vetvy do dvoch listov, vyvážime veľmi jednoducho – odtrhneme jablká z ťažšieho listu tak, aby sme počet jabĺk v oboch listoch vyrovnali.

Predstavme si teraz, že ideme vyvažovať uzol u, pričom predpokladáme, že všetky uzly jeho podstromu okrem samotného u sú už vyvážené. Pri vyvažovaní u už vo všeobecnosti nevieme jablká trhať po jednom, tak ako to bolo pri listoch, pretože potrebujeme udržať vyvážené oba jeho podstromy. Potrebujeme nájsť najväčší počet jabĺk, ktorý vieme dosiahnuť aj v jednom, aj v druhom podstrome.

Definujme si *výšku uzla* ako (maximálnu) výšku jablone nad daným uzlom, teda smerom k listom. (Každý list teda má výšku 0.) Dokážeme si nasledovné pozorovanie: ak má vrchol výšku h a jeho podstrom je vyvážený, tak z toho podstromu vieme bez porušenia vyváženosti trhať práve a len po 2^h jablkách. Toto dokážeme matematickou indukciou podľa h.

1° Vrchol s výškou 0 je list, z neho vieme jablká trhať po jednom.

2° Nech je celý podstrom nad uzlom v vyvážený a nech má uzol v výšku $h_v > 1$. Z neho idú dohora vetvy do dvoch vrcholov l a r, ktoré majú výšky h_l a h_r. Bez ujmy na všeobecnosti, nech $h_l \geq h_r$, a teda $h_v = h_l + 1$. Keďže v je vyvážený, podstromy s koreňmi l a r majú rovnako veľa jabĺk. Čiže ak nechceme pokaziť vyváženosť, musíme z nich trhať rovnako veľa. Zároveň ale musíme na každej strane jablká trhať tak, aby sme nepokazili vyváženosť daného podstromu. Z podstromu l vieme trhať po 2^{h_l} jablkách, z podstromu r po 2^{h_r}. Keďže $2^{h_r} \mid 2^{h_l}$, všetky počty, ktoré vieme odtŕhať aj v jednom, aj v druhom podstrome, sú tvaru $k \cdot 2^{h_l}$ pre $k \geq 0$. A teda z celého podstromu s koreňom v vieme trhať práve po $2 \cdot 2^{h_l} = 2^{h_v}$ jablkách.

Keď teda máme vyvážený strom s koreňom v, ktorý momentálne obsahuje j jabĺk, vieme, že všetky prípustné počty jabĺk v tomto strome sú práve tvaru $j - k \cdot 2^{h_v}$ pre $k \geq 0$ (pričom samozrejme nemôžeme ísť do mínusu). Na tomto vieme založiť riešenie s lineárnou časovou zložitosťou: postupne od listov ku koreňu spracúvame celý strom, pričom vždy, keď spájame dva podstromy, nájdeme najväčší počet jabĺk dosiahnuteľný na oboch stranách. Pri tom pomôže uvedomiť si a dokázať, že j je vždy násobkom 2^{h_v}. Pri implementácii takéhoto riešenia si ale treba dať pozor na to, že mocniny dvoch veľmi rýchlo rastú, a tak nám môže pre hlboké stromy pretiecť nejaká premenná.

My si ale ukážeme ešte trochu viac o vyvážených jabloniach a to nám potom umožní pohodlnejšiu implementáciu. Nové, silnejšie pozorovanie vyzerá nasledovne: Pre každú vyváženú jabloň platí: ak je na nej dokopy j jabĺk, tak pre každé i platí, že v každom liste, ktorý je vo vzdialenosti i od koreňa, je presne $j/2^i$ jabĺk.

Prečo je to tak? Predstavte si, že všetky jablká z celej vyváženej jablone presunieme do koreňa. Ich počet označíme j. Následne ich necháme, nech sa všetky naraz začnú presúvať do listov, do ktorých patria. V každom uzle, vrátane koreňa, sa jablká rozdelia na dve rovnako veľké polovice. Teda keď sú práve vo vzdialenosti i od koreňa, v každom vrchole ich je $j/2^i$. V tých vrcholoch, ktoré sú listami, tieto jablká zostanú, tie v uzloch sa opäť rozdelia na polovice a putujú ďalej.

Inými slovami: Nech h je výška celej našej jablone. Z práve dokázaného pozorovania vieme, že v každom liste, ktorý je v tejto výške, bude po vyvážení rovnaký počet jabĺk, označme ho x. Toto jediné x popisuje celú vyváženú jabloň

– každý list vo vzdialenosti i od koreňa bude mať $x \cdot 2^{h-i}$ jabĺk, a dokopy na celej jabloni bude presne $x \cdot 2^h$ jabĺk.

Hľadáme teda najväčšiu hodnotu našej neznámej x, ktorú vieme dosiahnuť pre dané súčasné rozloženie jabĺk. Pre každý list dostávame jedno obmedzenie hodnoty x zhora: ak je daný list vo vzdialenosti d_i od koreňa, mal by mať $x \cdot 2^{h-d_i}$ jabĺk. No keďže ich má v súčasnosti j_i a môžeme jablká len odoberať, nie pridávať, musí platiť $x \cdot 2^{h-d_i} \leq j_i$, a teda $x \leq \lfloor j_i / 2^{h-d_i} \rfloor$.

Stačí teda raz prejsť celým stromom, pre každý list zistiť jeho vzdialenosť od koreňa, z toho spočítať najväčšie prípustné x, a z toho rovno vieme výsledný počet jabĺk po vyvážení, a teda aj počet potrebných odtrhnutí. Aj toto riešenie má časovú zložitosť lineárnu od počtu vrcholov jablone, vieme ho však implementovať výrazne stručnejšie a navyše nám pri našej implementácii nehrozí pretečenie premenných.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
using namespace std;

vector< pair<int, int> > dohora; // pre každý uzol: kam z neho vedú dohora vetvy
vector<long long> pocet_jablk;   // pre každý list: koľko je v ňom jabĺk?
vector<bool> je_list;            // pre každý uzol: je to list?
vector<int> vzdialenosti;        // pre každý uzol: ako je ďaleko od koreňa?

void dfs(int kde, int vzdialenost) { // v akom vrchole som, ako ďaleko od koreňa
    vzdialenosti[kde] = vzdialenost;
    if (!je_list[kde]) {
        dfs( dohora[kde].first,  vzdialenost+1 );
        dfs( dohora[kde].second, vzdialenost+1 );
    }
}

int main() {
    int N; cin >> N;

    dohora.resize(N+1); pocet_jablk.resize(N+1); je_list.resize(N+1,false);

    for (int n=1; n<=N; ++n) {
        string typ; cin >> typ;
        if (typ=="U") {
            cin >> dohora[n].first >> dohora[n].second;
        } else {
            je_list[n] = true;
            cin >> pocet_jablk[n];
        }
    }

    vzdialenosti.resize(N+1);
    dfs(1,0);

    int h = *max_element( vzdialenosti.begin(), vzdialenosti.end() );
```

```
    long long x = *max_element( pocet_jablk.begin(), pocet_jablk.end() );
    for (int n=1; n<=N; ++n)
        if (je_list[n]) x = min(x, pocet_jablk[n] >> (h-vzdialenosti[n]) );

    long long zostane = x;
    if (zostane > 0) zostane <<= h; // nepretiečie, lebo ak x>0, h je nutne malé
    long long vsetky = accumulate(pocet_jablk.begin(),pocet_jablk.end(),0LL);
    cout << (vsetky - zostane) << "\n";
}
```

A-II-3 Zaokrúhľovanie

Táto úloha patrila medzi ťažšie z tohto kola Olympiády. Najskôr si ukážeme zopár dôležitých pozorovaní, následne si našu úlohu prevedieme na úplne ináč vyzerajúcu úlohu, a tú nakoniec vyriešime.

Dôležité pozorovania:

Prvé, čo si musíme uvedomiť, je, že celé časti našich čísiel sú úplne zbytočné. Ak máme niekde v mriežke necelé číslo x, môžeme ho pokojne nahradiť číslom $x - \lfloor x \rfloor$. Ešte stále dostaneme platnú tabuľku (súčet riadku aj stĺpca sme zmenili o celé číslo) a na riešení to nič nezmení – to zaokrúhlenie, ktoré fungovalo predtým, bude nutne fungovať aj teraz a naopak. Preto nám stačí vedieť riešiť úlohu, v ktorej sú všetky čísla na vstupe z intervalu $(0, 1)$.

Zoberme si teraz nejaký riadok a povedzme si, že jeho súčet je k. Keďže naše čísla ležia v intervale $(0, 1)$ pri zaokrúhľovaní sa rozhodujeme vždy medzi číslom 0 a 1. A ak máme zachovať súčet riadku k, znamená to, že práve k čísiel v tomto riadku musíme zmeniť na 1. Súčet každého riadku/stĺpca nám teda určuje, koľko čísiel v ňom máme zmeniť na 1.

Prevod na inú úlohu:

To čo sme zistili je síce pekné, ale uvažovať o tom v takomto tvare je veľmi obtiažne. Bolo by lepšie si to previesť na inú úlohu, o ktorej by sa nám uvažovalo ľahšie. K tomu nám môže pomôcť, že keď zmeníme nejaké číslo v mriežke na 1, ovplyvní to jeden riadok a jeden stĺpec. Hľadáme teda niečo čo dáva do súvisu dve množiny vecí.

Vhodná štruktúra je bipartitný graf. Budeme mať dve množiny vrcholov R a S, kde každý vrchol v R predstavuje jeden riadok a vrchol v S stĺpec. Hrana medzi dvoma vrcholmi r_i a s_j znamená, že sme číslo v i-tom riadku a j-tom stĺpci zaokrúhlili na 1. Zo vstupu vieme počet stĺpcov aj riadkov. A vieme z neho pre každý riadok aj stĺpec zistiť, koľko čísiel v ňom sa má zaokrúhliť na 1 – teda počet hrán, ktoré musia vychádzať z konkrétneho vrcholu v našom grafe.

Dostávame teda úlohu z teórie grafov. Máme zostrojiť bipartitný graf, pričom každý vrchol má vopred určený počet hrán, ktoré z neho majú vychádzať – stupeň daného vrchola.

(Odbočka: Uvedomte si, že táto nová úloha je o niečo všeobecnejšia. Totiž existujú niektoré jej zadania, ktoré nezodpovedajú žiadnej matici reálnych čísel. To nám ale nevadí – ak túto novú úlohu vyriešime, vyriešime tým aj našu pôvodnú.)

Problémom je, že nemôžeme mať násobné hrany, teda nemôže byť medzi dvoma vrcholmi viac ako jedna hrana. Ak by sme teda priraďovali hrany len tak ako príde, mohlo by sa nám stať, že síce riešenie existuje, ale my ho nenájdeme – na konci nám zostanú nejaké vrcholy, ktoré ešte nemajú dostatočný stupeň, no zároveň už nemôžeme pridať žiadnu hranu, lebo všetky hrany, ktoré by sme potrebovali pridať, už v grafe máme.

Potrebujeme nájsť nejaký spôsob zostrojovania grafu, ktoré niečo takéto vylúči – teda ktorý nám zaručí, že ak graf s danými stupňami vrcholov existuje, tak jeden taký nájdeme.

Pažravý algoritmus:

Vyslovíme teraz tvrdenie, ktoré nám pomôže nájsť efektívne riešenie našej novej úlohy.

Tvrdíme: Ak existuje bipartitný graf B s danými stupňami vrcholov, existuje aj B', ktorý má rovnaké stupne vrcholov a navyše platí, že konkrétny vrchol r z množiny R je spojený s $\deg(r)$ vrcholmi z S, ktoré majú najväčší stupeň.[1]

Dôkaz: Majme teda nejaký bipartitný graf B a v ňom vrchol r. Ukážeme, že graf B vždy vieme postupne prerobiť na požadovaný graf B'

Ak je r pripojený ku $\deg(r)$ vrcholom s najvyšším stupňom, je všetko v poriadku a $B' = B$. Ak nie, tak existuje vrchol s_1 v množine S, ktorý je spojený s r, ale nepatrí medzi $\deg(r)$ najväčších. A tiež existuje vrchol s_2 z množiny S, ktorý leží medzi $\deg(r)$ najväčšími, ale nie je spojený s r. Keďže $\deg(s_2) > \deg(s_1)$, tak existuje vrchol r' z množiny R, ktorý je spojený s s_2, ale nie s s_1. Ak teraz nahradíme hrany $r - s_1$ a $r' - s_2$ za hrany $r - s_2$ a $r' - s_1$, máme stále korektný bipartitný graf, opravili sme si však jednu zlú hranu z vrcholu r. Opakovaním tohto postupu prerobíme v konečnom počte krokov pôvodný graf B na graf, v ktorom je r spojený s $\deg(r)$ vrcholmi s najväčším stupňom, q.e.d.

Toto nám dáva postup ako budovať daný graf: Zoberieme ľubovoľný $r \in R$ a nájdeme mu zodpovedajúce vrcholy v S. Teraz môžeme na vrchol r zabudnúť

[1] Značenie $\deg(r)$ predstavuje stupeň vrchola v.

a vrcholom z S, ktoré sme s ním spojili, znížiť stupeň o 1. Tým dostávame ten istý problém, len na menšom grafe. Tento postup teda opakujeme, až kým buď nespracujeme celé R, alebo nenarazíme na spor.

Našu pôvodnú úlohu teda vyriešime nasledovne: Pre každý riadok a stĺpec zistíme počet jednotiek, ktoré v ňom musíme pridať – teda stupeň zodpovedajúceho vrcholu v našom grafe. Následne budeme postupne vypĺňať jednotlivé riadky. (Keďže nezáleží na poradí, v akom to robíme, budeme ich jednoducho vypĺňať v poradí, v akom sú na vstupe.) Nech teraz spracúvame riadok, v ktorom má byť k jednotiek. Na to nám stačí nájsť k stĺpcov, v ktorých chýba najviac jednotiek, a vyplniť ich tam.

Jediná otvorená otázka je, ako budem vyberať k stĺpcov s najväčším stupňom. Použitím haldy alebo nejakého klasického efektívneho algoritmu (napr. MergeSortu alebo knižničnej funkcie `sort`) na triedenie dostanem pre tabuľku rozmerov $r \times s$ časovú zložitosť $O(rs \log s)$. Uvedomme si však, že čísla, ktoré triedim, sú z rozsahu 0 až r. Preto môžem použiť CountSort, alebo iný triediaci algoritmus s lineárnou časovou zložitosťou. Takto dostaneme časovú zložitosť $O(rs)$. Toto je zároveň aj optimum, keďže musíme načítať vstup.

Listing programu (C++)

```
#include<cstdio>
#include<algorithm>
#include<vector>
#include <cmath>
using namespace std;

int main(){
    int r,s;
    scanf("%d %d ",&r,&s);
    double A[r][s];
    for(int i=0; i<r; i++)
        for(int j=0; j<s; j++)
        {
            double p;
            scanf("%lf ",&p);
            A[i][j]=p-floor(p);
        }
    int R[r],S[s];
    char V[r][s];
    for(int i=0; i<r; i++)
    {
        double p=0;
        for(int j=0; j<s; j++) p+=A[i][j];
        R[i]=ceil(p);
    }
    for(int i=0; i<s; i++)
    {
        double p=0;
        for(int j=0; j<s; j++) p+=A[j][i];
        S[i]=ceil(p);
    }
```

```
    for(int i=0; i<r; i++)
        for(int j=0; j<s; j++) V[i][j]='D';
    vector<vector<int> > C;
    for(int i=0; i<r; i++)
    {
        C.clear();
        C.resize(s+1);
        for(int j=0; j<s; j++) C[S[j]].push_back(j);
        int p=0;
        int kde=s;
        while(p!=R[i])
        {
            for(int k=0; k<C[kde].size(); k++)
            {
                if(p==R[i]) break;
                V[i][C[kde][k]]='H';
                p++; S[C[kde][k]]--;
            }
            kde--;
        }
    }
    for(int i=0; i<r; i++)
    {
        for(int j=0; j<s; j++) printf("%c",V[i][j]);
        printf("\n");
    }
}
```

Alternatívne riešenie:

Súťažná úloha sa dala (síce pomalšie, ale ešte stále v polynomiálnom čase) vyriešiť aj bez transformácie na úlohu o zostrojení bipartitného grafu. Toto pomalšie riešenie uvádzame aj z toho dôvodu, že z neho vyplynie, že naša úloha vždy má riešenie – teda že vždy existuje aspoň jeden vhodný spôsob zaokrúhľovania.

V tomto riešení budeme zaokrúhľovať čísla postupne. Vždy si nájdeme nejakú množinu čísel, ktoré ešte nie sú celé, a vhodne ju upravíme. Vhodná úprava bude mať dve vlastnosti: nezmení súčet v žiadnom riadku ani stĺpci, a navyše po nej bude aspoň jedno zo zmenených čísel celé.

Najjednoduchšia úprava bude vyzerať nasledovne: Vyberieme si dva riadky a dva stĺpce také, že všetky štyri čísla, ktoré ležia v tých riadkoch a zároveň v tých stĺpcoch sú necelé. Teraz môžeme dve z nich („ľavé horné“ a „pravé dolné“) o nejakú hodnotu δ zvýšiť a zároveň zvyšné dve o tú istú hodnotu δ znížiť. Tým sa zjavne nezmení súčet žiadneho riadku ani stĺpca. No a keď hodnotu δ zvolíme vhodne, niektoré z nich (aspoň jedno) stúpne na 1 alebo klesne na 0, zatiaľ čo ostatné (najviac tri) ostanú v intervale $(0, 1)$.

Napríklad keby sme niekde v tabuľke videli čísla:

```
.... 0.47 .... 0.21 ....
.... .... .... .... ....
.... 0.17 .... 0.62 ....
```

tak si zvolíme $\delta = 0.17$, čím dostaneme:

```
.... 0.64 .... 0.04 ....
.... .... .... .... ....
.... 0.00 .... 0.79 ....
```

a úspešne sme tak zvýšili počet celých čísel.

Samozrejme, časom sa nám takéto štvorice čísel minú – v každej už bude aspoň jedno celé číslo. Vtedy budeme musieť začať hľadať väčšie skupiny čísel, ktoré zmeníme.

Ako ich hľadať? Ak ešte nie sme hotoví, je v našej tabuľke aspoň jedno necelé číslo. Jedno také si vyberieme. Riadok, v ktorom je, má celočíselný súčet, preto v ňom musí byť aspoň jedno ďalšie necelé číslo. Jedno také si nájdeme a pozrieme sa pre zmenu na jeho stĺpec. Aj v tom musí niekde byť ešte aspoň jedno iné necelé číslo. A takto pokračujeme ďalej, striedavo hľadajúc ďalšie číslo v riadku a v stĺpci.

A keďže pokračovať môžeme do nekonečna, časom sa nám nejaký riadok alebo stĺpec zopakuje. V tom okamihu sme našli cyklus: nejakú postupnosť k riadkov a k stĺpcov, ktorá nám určuje $2k$ necelých čísel – v každom riadku aj stĺpci dve z nich. No a s nimi spravíme presne to isté: na striedačku ich budeme zväčšovať a zmenšovať o tú istú hodnotu δ.

Práve sme teda ukázali, že vždy, keď ešte máme v našej tabuľke nejaké necelé čísla, vieme ju upraviť tak, aby sa aspoň jedno z nich zmenilo na celé. Opakovaním vyššie popísaného postupu teda časom musíme dostať tabuľku, v ktorej už sú všetky čísla celé.

Hľadanie jednej vhodnej skupiny políčok vieme v tabuľke rozmerov $r \times s$ spraviť v čase $O(rs)$ – lebo pred nájdením cyklu prejdeme každý riadok a každý stĺpec najviac raz. Jej úpravu potom spravíme v zanedbateľnom čase $O(\min(r, s))$. No a keďže máme tabuľku s rs políčkami a v každej iterácii zmeníme na celé aspoň jedno z nich, bude nám určite stačiť rs iterácií. Celkovú časovú zložitosť tohto riešenia teda vieme zhora odhadnúť ako $O(r^2 s^2)$.

A-I-4 Log-space výpočty

Podúloha A – Násobenie veľkých čísel:

Násobiť veľké čísla sme sa učili už na základnej škole. Čísla α a β si jednoducho napíšeme pod seba, potom postupne násobíme α všetkými ciframi β (na to stačí poznať malú násobilku), pričom pri každej ďalšej cifre sa posunieme o jeden stĺpec doľava. Nakoniec všetky medzivýsledky sčítame. Napríklad pre $\alpha = 137$ a $\beta = 123$ vyzerá násobenie takto:

$$\begin{array}{cccccc} & & & 1 & 3 & 7 \\ & & \times & 1 & 2 & 3 \\ \hline & & & 4 & 1 & 1 \\ & & 2 & 7 & 4 & \\ & 1 & 3 & 7 & & \\ \hline 1 & 6 & 8 & 5 & 1 & \end{array}$$

Tento algoritmus beží v čase $O(n^2)$ a dá sa jednoducho implementovať s pomocným poľom. Pri log-space výpočtoch si toto nemôžeme dovoliť a musíme sa zamyslieť nad implementáciou *bez* pomocného poľa.

Výsledok budeme počítať cifru po cifre, stĺpec po stĺpci. Pre jednoduchosť zatiaľ zabudnime na prenosy do vyšších rádov. Potom posledná cifra $C[0]$ je zrejme $A[0] * B[0]$.

$$\begin{aligned} C[1] &= A[0] * B[1] + A[1] * B[0] \\ C[2] &= A[0] * B[2] + A[1] * B[1] + A[2] * B[0] \\ C[3] &= A[0] * B[3] + A[1] * B[2] + A[2] * B[1] + A[3] * B[0] \\ &\vdots \end{aligned}$$

Vo všeobecnosti je $C[i]$ súčet nejakých $A[j]*B[k]$, pričom $j+k = i$. V skutočnosti k tomuto môže ešte pribudnúť prenos z nižšieho rádu a ak dostaneme výsledok > 9, $C[i]$ bude iba posledná cifra tohto súčtu. Teda

$$\begin{aligned} \text{súčet} &\leftarrow \text{prenos} + \sum_{j+k=i} A[j] * B[k] \\ C[i] &\leftarrow \text{súčet mod } 10 \\ \text{prenos} &\leftarrow \lfloor \text{súčet}/10 \rfloor \end{aligned}$$

Takto stačí použiť len zopár pomocných premenných. Navyše súčet bude nadobúdať len hodnoty od 0 po $90n$ (môžeme mať najviac n-krát $9 * 9 = 81n$ plus prenos $9n$ z nižšieho rádu), takže program je naozaj log-space.

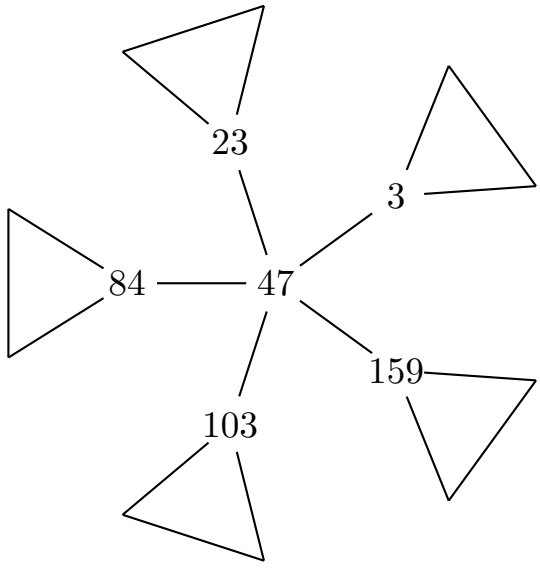
Listing programu (Pascal)

```
var n : integer;                          { vstup: dĺžka čísel }
    A, B : array [0..n-1] of integer;     { vstup: alfa, beta }
    C : array [0..2*n-1] of integer;      { výstup: gama = alfa * beta }
    i, j, p : integer;

begin
    p := 0;
    for i := 0 to 2*n-1 do begin
        for j := max(0,i+1-n) to min(i,n-1) do
            p := p + A[j] * B[i - j];
        C[i] := p mod 10; { posledná cifra je C[i] }
        p := p div 10;    { zvyšok sa prenesie do vyššieho rádu }
    end;
end.
```

Podúloha B – Prehľadávanie lesu:

Tradične by sme túto úlohu riešili prehľadávaním do hĺbky alebo do šírky. Začneme z jedného vrcholu, prehľadáme celý jeho komponent a ak pritom natrafíme na druhý vrchol, sú spolu, inak nie. Obe riešenia však používajú veľa pamäte – pre každý vrchol si pamätáme, či sme ho už navštívili alebo nie a navyše máme zásobník alebo frontu vrcholov, ktoré sa chystáme navštíviť. V log-space programoch si však môžeme pamätať informácie iba o zopár (konštantne veľa) vrcholoch.



Využijeme, že zadaný graf je les. Predstavme si, že stojíme vo vrchole 47 na

obrázku vyššie. Okolo nás je 5 susedov a keby sme vrchol 47 odstránili, graf by sa rozpadol na 5 podstromov (zakreslených len schématicky ako trojuholníky). Nemáme dosť pamäte na to, aby sme si pamätali, v ktorých podstromoch sme už boli a v ktorých nie, ale môžeme si susedov pekne zoradiť podľa identifikátorov. Budeme postupovať pomocou jednoduchého pravidla, pričom si budeme pamätať dve veci: v ktorom vrchole sme a z ktorého vrcholu práve prichádzame. Pokračovať budeme vždy cyklicky tým nasledujúcim vrcholom v utriedenom poradí.

Napríklad, ak sme do 47 prišli z vrcholu 84, ďalej budeme pokračovať vo vrchole 103 (pričom si pamätáme, že sme tam prišli zo 47). Keď sa prehľadávanie z vrcholu 103 vráti (t.j. sme v 47 a prišli sme zo 103), nasledujúci vrchol bude 159. Keď sa vrátime zo 159, ideme do 3, atď.

Treba si ešte rozmyslieť odpovede na tri otázky:

Prvá otázka: Funguje takéto prehľadávanie? Naozaj zaručene prejdeme celý strom? To sa dá dokázať napríklad indukciou: Ak vôjdeme do listu, tak z neho po tej istej hrane výjdeme aj naspäť (lebo inú hranu nemáme). Ak stojíme vo vrchole w a jeho susedia sú $w_1, w_2, \ldots, w_k$ zoradení od najmenšieho po najväčšieho, pričom do w sme sa dostali z w_j, potom budeme pokračovať vo w_{j+1}. Z indukčného predpokladu sa do w z w_{j+1} vrátime a budeme pokračovať vo w_{j+2}, atď., až kým cyklicky neobehneme všetkých susedov a vrátime sa do w_j.

Celý algoritmus si môžeme predstaviť aj tak, že vrcholy grafu sú miestnosti a hrany sú chodby. My sa pravou rukou držíme jednej steny a ideme len popri nej. Ak je daný graf strom, potom ho takýmto spôsobom celý prejdeme. (Rozmyslite si, že pre všeobecné grafy to nefunguje.)

Druhá otázka: Kedy máme zastať? Jedna možnosť je zapamätať si, v ktorom vrchole sme začínali a ktorá bola prvá hrana, ktorou sme sa vydali (to sú len 2 čísla). Ak sa dostaneme na začiatok a budeme sa chcieť vydať toutou hranou, môžeme skončiť. Druhá, lenivá možnosť je spraviť jednoducho $2m$ krokov – totiž komponent môže mať najviac m hrán a keď každú prejdeme dvakrát (tam a späť), dostaneme sa na začiatok.

Tretia otázka: Dá sa tento algoritmus implementovať ako log-space program? Dá. Pamätáme si iba v ktorom vrchole sme a odkiaľ sme prišli. V danom vrchol vždy prejdeme celý zoznam hrán a nájdeme nasledujúceho suseda (to je najmenší väčší alebo, ak väčší sused neexistuje, tak úplne najmenší sused; toto je implementované vo funkcii `dalsia_hrana`).

Pre zaujímavosť na záver ešte poznamenajme, že naše riešenie dosť podstatne využíva fakt, že graf na vstupe je les a komponenty sú stromy. Log-space program pre všeobecné neorientované grafy existuje, avšak je oveľa oveľa kom-

plikovanejší (nad rámec OI aj vysokoškolského učiva); pred necelými 9 rokmi ho vynašiel Omer Reingold. Podobná otázka pre orientované grafy, teda či existuje algoritmus s malou pamäťovou zložitosťou, ktorý by rozhodol, či sa v danom grafe dá z vrcholu u dostať do vrcholu v, je dodnes otvorená. Poznáme iba algoritmus s pamäťovou zložitosťou $O((\log n)^2)$ a predpokladá sa, že žiadny log-space program ani neexistuje. Nikto to však zatiaľ nevie dokázať.

Listing programu (Pascal)

```
var n, m, u, v : integer;           { vstup: počet vrcholov a hrán, štart a cieľ }
    A, B : array [1..m] of integer; { vstup: hrany grafu }
    spolu : integer;                { výstup: sú u+v v tom istom komponente? }
    w, z, i, tmp : integer;         { som vo vrchole w, prisiel som sem zo z }

function dalsia_hrana (w, z : integer) : integer;
var i, minn, nextn : integer;
begin
    minn := z; { sused s najmenším číslom }
    nextn := m+1; { sused s najmenším číslom väčším ako z }
    for i := 1 to m do begin
        if A[i] = w then begin
            if B[i] < minn then minn := B[i];
            if (B[i] > z) and (B[i] < nextn) then nextn := B[i];
        end;
        if B[i] = w then begin
            if A[i] < minn then minn := A[i];
            if (A[i] > z) and (A[i] < nextn) then nextn := A[i];
        end;
    end;
    { ak neexistuje sused s väčším číslom, vrátime najmenšieho suseda }
    if nextn <= m then dalsia_hrana := nextn
    else dalsia_hrana := minn;
end;

begin
    w := u
    z := -1;
    for i := 1 to 2*m + 47 do begin
        if w = v then begin
            spolu := 1; halt;
        end;
        tmp := dalsia_hrana (w, z);
        z := w;
        w := tmp;
    end;
    spolu := 0;
end.
```

Riešenia krajského kola kategórie B

B-II-1 Nutela

Tí, ktorí ste nevymysleli vzorové riešenie, nemali by ste hneď házdať flintu do žita. Za 3 body stačilo napísať riešenie, ktoré skontroluje všetky možné dvojice téglikov.

No keďže 3 body nie sú veľa, radšej si ukážeme, ako bez veľkej námahy vylepšiť toto riešenie na celých 8 bodov. Nebudeme skúšať dvojice, ale skúsime si postupne vybrať každý téglik ako prvú nutelu z dvojice a následne druhý téglik nejako, čo najrýchlejšie dohľadať.

Keď máme zvolený prvý téglik, vieme jeho veľkosť, označme si ju c. Druhý téglik teda musí mať veľkosť od $a - c$ po $b - c$, a hľadáme ho v usporiadanom poli. To vieme robiť binárnym vyhľadávaním, podobne, ako keď hľadáme meno v telefónnom zozname, alebo hádame, aké číslo si myslí kamarát, či nebodaj hľadáme poklad s pomocou vysávača (viď úlohu 3).

Keďže binárne vyhľadávanie má časovú zložitosť $O(\log n)$, celý algoritmus bude bežať v čase $O(n \log n)$. Pamäťvá zložitosť bude $O(n)$.

Vzorové riešenie:

Vieme, že ak tam niekde správne nutely sú, budú v intervale od prvého po posledný téglik (vrátane). (Kde inde by mohli byť?) Počas algoritmu sa budeme tento interval snažiť zmenšovať, až kým buď nenájdeme správnu dvojicu, alebo neskončíme s intervalom dĺžky 1 a vypíšeme „nedá sa“.

Stačí dokola opakovať nasledujúce zmenšovanie.

Majme nejaký rad $m > 1$ utriedených téglikov s veľkosťami $x_1 < x_2 < \cdots < x_m$. Potom môžu nastať tri možnosti.

- Buď prvý a posledný téglik majú dobrý súčet ($a \leq x_1 + x_m \leq b$), vtedy vypíšeme ich hodnoty a skončíme.
- Môžu mať aj primalý súčet ($x_1 + x_m < a$). Vtedy vieme, že prvý z téglikov určite nebude patriť do správnej dvojice. Totiž ak uvažujeme dvojicu x_1 a x_i pre nejaké i, tak vieme, že $x_1 + x_i \leq x_1 + x_m < a$, čo je príliš málo na to, aby to bola správna dvojica. Preto môžeme prvý téglik zahodiť a zaujímať sa len o zvyšných $m - 1$.
- Posledná možnosť je, že prvý a posledný téglik majú priveľký súčet ($x_1 + x_m > b$). Veľmi podobnou úvahou zistíme, že môžeme z intervalu uvažovaných téglikov vynechať posledný téglik.

Pokiaľ sa na polici žiadna vyhovujúca dvojica téglikov nenachádzala, skončíme časom s jediným téglikom a v tej chvíli už môžeme dať zápornú odpoveď.

Časová zložitosť bude linárna od počtu téglikov, $O(n)$, pretože v každom kroku spravíme nejaké výpočty, ktoré prebehnú v konštantnom čase, a následne ak sme ešte neskončili, tak jeden téglik zahodíme. No a dokopy môžeme zahodiť nanajvýš $n - 1$ téglikov. Rýchlejšie sa túto úlohu už nedá, lebo musíme aj tak načítať celý vstup. (Čo keby správna dvojica bola až na konci?)

Pamäťová zložitosť bude tiež $O(n)$, pretože si uchovávame všetky tégliky naraz v pamäti. (Zlepšiť to takisto nevieme, lebo v okamihu, keď načítavame posledný téglik, potrebujeme si pamätať ostatné. Keby sme si nejaký nepamätali, môže sa stať, že jediná správna dvojica bude posledný téglik a ten, ktorý si nepamätáme.)

Listing programu (Pascal)

```
var n,a,b:longint;
    i,zac,kon:longint;
    x:array[1..1000000] of longint;
begin
    read(n,a,b);
    for i := 1 to n do read(x[i]);
    zac := 1;
    kon := n;
    while kon-zac>0 do begin
        if x[zac] + x[kon] < a then begin
            inc(zac);
            continue;
        end;
        if x[zac] + x[kon] > b then begin
            dec(kon);
            continue;
        end;
        writeln(x[zac],' ',x[kon]);
        exit;
    end;
    writeln('nedá sa');
end.
```

B-II-2 Pošta

Tento vzorák, vás dúfam naučí mnohým fintám pre pravých chlapov a pravé ženy. Najskôr si ukážeme, ako sa dá na strome hľadať cesta medzi dvoma vrcholmi a následne si ukážeme, že vo vzorovom riešení tie cesty ani nepotrebujeme. Prekvapivé? Tak to si ešte počkajte. A samozrejme, keďže sme pravé ženy a praví chlapi, nebudeme sa okúnať s metaforami ako domy a ulice, ale ako sa patrí v teórii grafov, domy nazveme vrcholy a ulice hrany v našom grafe.

Prehľadávanie do hĺbky:

Prvým problémom, ktorý riešime v tejto úlohe, je hľadanie cesty medzi dvoma vrcholmi. Ak by sme toto vedeli, hneď by sme mali triviálne riešenie – pre každú dvojicu vrcholov nájdeme cestu a jej dĺžku zarátame do výsledku. Ako však nájsť cestu medzi vrcholmi x a y?

V domácom kole sme sa zoznámili s prehľadávaním do šírky, čo bol jednoduchý algoritmus na hľadanie najkratšej cesty v neohodnotenom grafe. Tu pracujeme tiež s neohodnoteným grafom a preto sa tento algoritmus dá použiť. Náš graf je však strom a so stromami je oveľa viac spätý algoritmus prehľadávania do hĺbky (skrátene DFS z anglického Depth-First Search). Ukážeme si teda, ako tento algoritmus funguje, neskôr sa nám totiž ešte zíde pri lepšom riešení.

DFS si zakladá na rekurzívnom vnáraní čoraz hlbšie a hlbšie. Presnejšie, nech začneme v nejakom vrchole x. Tento vrchol si označíme ako navštívený. Postupne sa budeme pozerať na všetkých susedov x a vždy, keď nájdeme nejakého, ktorý je zatiaľ nenavštívený, rekurzívne sa vnoríme – teda označíme ten vrchol za navštívený a začneme rovnakým spôsobom prezerať jeho susedov. Ak sme takto spracovali všetkých susedov aktuálneho vrcholu, vrátime sa v rekurzii späť. Konkrétne, algoritmus sa dá jednoducho implementovať nasledovným spôsobom.

Listing programu (C++)

```
vector< vector<int> > G; // V[x] je zoznam vrcholov, ktoré susedia s vrcholom x
bool T[n];               // T[x] je true ak som už vrchol x navštívil
int depth = -1;          // hĺbka pod začiatočným vrcholom, v ktorej práve sme

void dfs(int v) {
    ++depth;               // ideme hlbšie
    T[v]=true;             // vrchol v je odteraz navštívený
    for(int i=0; i<G[v].size(); i++) {
        int w=G[v][i];     // vrchol w je susedom vrcholu v
        if(T[w]) continue; // ak už bol navštívený, nezaujíma nás
        dfs(w);     // inak rekurzívne ofarbíme w, susedov w, ich susedov, atď.
    }
    --depth;        // a keď sme všetko prezreli, vynoríme sa o úroveň vyššie
}
```

Vidíme, že je to naozaj elegantné. A ako nám to pomôže v našom probléme? Vidíme, že v programe mám premennú `depth`. Tá označuje, ako ďaleko sme sa dostali od začiatočného vrcholu. Vždy keď sa vnoríme hlbšie do rekurzii, premennú zväčším, znamená to, že sme sa od x vzdialili o ďalšiu hranu, keď sa z rekurzie vynárame, premennú zmenšíme, sme o hranu bližšie. To ale znamená, že ak začneme prehľadávanie z vrcholu x, keď objavíme vrchol y, tak v premennej `depth` máme jeho vzdialenosť od vrchola x. A to je presne to, čo sme potrebovali.

Časová zložitosť prehľadávania do hĺbky vo všeobecnom súvislom grafe je $O(n+m)$, kde n je počet vrcholov a m počet hrán toho grafu. To preto, že rekurziu voláme len na neoznačený vrchol, takže funkciu `dfs()` zavoláme presne n-krát. A v rámci týchto volaní sa pozriem na každú hranu dvakát – z každého konca raz.

V strome platí, že jeho počet hrán m je presne rovný $n-1$. Preto je časová zložitosť prehľadávania do hĺbky na strome $O(n)$, teda lineárna od počtu vrcholov stromu.

Takto dostávame prvé riešenie našej pôvodnej úlohy: pre každú dvojicu (x, y) spustíme DFS z vrcholu x, zistíme vzdialenosť do y a tú pripočítame k výsledku. Takéto riešenie má časovú zložitosť $O(n^3)$.

Toto riešenie sa však dá jednoducho zlepšiť. Uvedomme si, že pri prehľadávaní do hĺbky navštívime každý vrchol práve raz. To znamená, že ak začneme prehľadávať z vrcholu x, tak postupne prejdeme všetky ostatné vrcholy. A teda vieme počas jediného prehľadávania zistiť vzdialenosť od vrcholu x ku všetkým ostatným. Takto ľahko dostaneme riešenie s časovou zložitosťou $O(n^2)$. Z každého vrcholu spustíme jedno prehľadávanie a k výsledku pripočítame vzdialenosti ku všetkým ostatným vrcholom. Za takéto riešenie by sme dostali slušných 7 bodov, my (ako praví chlapi a pravé ženy!) sa s tým však samozrejme neuspokojíme :-).

Vzorové riešenie:

Myšlienka vzorového riešenia bude vychádzať z myšlienky, ktorú sme použili v domácom kole pri počítaní súčtov všetkých možných úsekov postupnosti. Mimochodom, táto úloha je všeobecnejšou verziou dotyčnej úlohy domáceho kola: Predstavte si jednu rovnú cestu a na nej $n+1$ domov tak, že vzdialenosti medzi susednými domami sú jednotlivé čísla postupnosti. Potom každý úsek postupnosti zodpovedá vzdialenosti niektorých dvoch domov. Teraz teda riešime presne tú istú úlohu, len tentokrát môže mať dedina ľubovoľnú stromovú topológiu.

Pozrime sa teda na cesty bližšie, hlavne to, z čoho sa skladajú. No predsa z jednotlivých hrán. Náš strom ich má, ako už vieme, presne $n-1$. A každá hrana nejak prispieva k celkovému výsledku. Presnejšie, konkrétna hrana je vo výsledku zarátaná toľkokrát, koľkokrát cez ňu vedie nejaká cesta. Ak by sme vedeli zistiť tento počet, vedeli by sme úlohu ľahko vyriešiť. Pre každú hranu by sme zrátali, koľko ciest cez ňu prechádza, a sčítali týchto $n-1$ čísel.

Zoberme si nejakú hranu e. Ak ju z nášho stromu vyberieme, vidíme, že náš strom sa rozpadne na dve časti – komponenty. Toto v strome platí pre ľubovoľnú

hranu. Od čoho závisí, či cesta medzi vrcholmi x a y, prechádza cez hranu e? Ak vrcholy x a y ležia v tom istom komponente, cez hranu e ich spoločná cesta neprechádza, lebo existuje spojenie týchto vrcholov v rámci toho komponentu. No a naopak, ak x a y ležia v rôznych komponentoch, vtedy cesta medzi nimi musela prechádzať hranou e, lebo ona jediná spája tieto dva komponenty.

Ak si označíme počet vrcholov v prvom komponente p_a a v druhom komponente p_b, tak dostávame, že hranou e prechádza práve $p_a \cdot p_b$ ciest spájajúcich nejaké dvojice vrcholov. Bolo by teda fajn, keby sme pre každú hranu vedeli rýchlo zistiť hodnoty p_a a p_b. Dokonca nám stačí len jedna z nich, lebo vieme, že vždy platí $p_a + p_b = n$.

Vyberme si teraz nejaký vrchol x a zavesme náš strom za tento vrchol. Znamená to, že všetky ostatné visia pod ním. Odborne sa tento vrchol nazýva koreň. To znamená, že z každého vrchola trčí jedna hrana hore, smerom k x. Samozrejme okrem samotného x, keďže ten je najvyššie a máme len $n-1$ hrán. Každá hrana tento strom rozdelí na dva komponenty: jeden je tvorený spodným vrcholom hrany a všetkým, čo visí pod ním, druhý je celý zvyšok stromu. Keby sme pre pre každý vrchol zistili, koľko vrcholov visí pod ním (teda aký veľký je jeho podstrom), určovalo by nám to hodnotu p_a pre hranu vedúcu z neho dohora.

Pomocou DFS to nie je problém zistiť. Počas prehľadávania si budeme pre každý vrchol počítať, koľko vrcholov je pod ním. Toto zistím tak, že sčítam veľkosti podstromov všetkých jeho susedov, ktorý ležia nižšie ako on (čo sú vrcholy, do ktorých som sa z tohto vrchola rekurzívne vnáral) a pričítam 1 za ten samotný vrchol. Potom si k výsledku prirátam $p(n-p)$ kde p je veľkosť podstromu pre daný vrchol. Na konci teda dostanem žiadaný výsledok. Toto všetko zrátam počas jedného prehľadávanie, preto zložitosť tohto algoritmu je $O(n)$. Pamätať si potrebujem samotný graf, čo nám zaberie $O(n)$ pamäte.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

long long vys=0;
int n;
vector<vector<int> > G;
vector<bool> T;         // T[x] hovorí, či už bol vrchol x navštívený
vector<int> P;          // P[x] je veľkosť podstromu s koreňom vo vrchole x

void dfs(int v) {
    T[v]=true;
    for(int i=0; i<G[v].size(); i++) {
```

```
        int w=G[v][i];
        if(T[w]) continue;
        dfs(w);          // spracujeme všetky vrcholy v podstrome s koreňom w
        P[v]+=P[w];      // pridáme veľkosť podstromu s koreňom w k našej veľkosti
    }
    P[v]++;              // za samotný vrchol v
    vys+=(long long)P[v]*(n-P[v]);
}

int main() {
    // načítame vstup
    scanf("%d ",&n);
    G.resize(n); T.resize(n,false);
    P.resize(n,0);
    for(int i=0; i<n-1; i++) {
        int x,y;
        scanf("%d %d ",&x,&y);
        x--; y--;
        G[x].push_back(y);
        G[y].push_back(x);
    }
    // spustíme prehľadávanie do hĺbky
    dfs(0);
    printf("%lld\n",vys);
    return 0;
}
```

B-II-3 Poklad

Než sa pozrieme na náš príklad s vysávačom, zamyslime sa nad jednoduchšou hrou. Niekto si myslí číslo od 1 do n a my máme uhádnuť, ktoré to je. Keď skúsime nejaké číslo, dozvieme sa, či sme uhádli, alebo je správna hodnota väčšia, alebo menšia ako náš tip.

Ak na začiatku vieme, že správna hodnota je medzi 1 a 100, dobrá stratégia je tipnúť si 50. Buď sa dozvieme "viac" (takže zostane len 50 kandidátov na správnu hodnotu), alebo "menej" (iba 49 kandidátov), alebo, ak máme šťastie, to bude presne 50.

My sa samozrejme nechceme spoliehať len na šťastie, chceme stratégiu, čo bude dobrá aj v tom najhoršom prípade. Keby sme vyberali čísla, čo sú ďaleko od stredu, ľahko by nám mohlo zostať veľa kandidátov. Zatiaľ čo ak si vždy vyberieme číslo v strede zostávajúcich kandidátov, po každom ťahu ich zostane najviac polovica. Takže tejto stratégii vždy postačí $\lfloor \log_2 n \rfloor$ ťahov. Toto sa volá binárne vyhľadávanie (lebo hľadáme správnu hodnotu tým, že to vždy delíme binárne, čiže na dve polovice).

Podúloha A: s predlžovačkou:

Späť k vysávaču. To, čo by sme potrebovali, je vedieť zisťovať, či je poklad

vpravo alebo vľavo od nejakého políčka s (toho, čo je v strede úseku, kde sa ešte poklad môže nachádzať). Ale vieme dostať odpoveď len na otázku, či je nové vysávané políčko k pokladu bližšie alebo ďalej ako posledné vysávané políčko (označme ho p). Čo s tým? Spravíme to tak, že povysávame políčko $2s - p$, čiže to, ktoré je oproti p, keď sa pozeráme z políčka s. Podľa hrkania vysávača sa dozvieme, či je poklad bližšie ku p alebo ku $2s - p$. A keďže sú tieto dve políčka umiestnené symetricky okolo s, hrkanie nám takto povie na ktorej strane od s ten poklad je.

Máme $n \leq 10^9$, takže nám postačí povysávať $\lfloor \log_2 10^9 \rfloor + 1 = 30$ políčok. To je pod limitom 35. (To $+1$ je prvé povysávané políčko. Nezáleží, ktoré si vyberieme – nič sa z neho nedozvieme, a v ďalšom ťahu bez ohľadu na p vieme ťahať tak, ako potrebujeme.)

Listing programu (C++)

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

int main() {
  string odpoved;
  int n;
  cin >> n;

  int p = 1;
  cout << "vysavaj " << p << endl;
  cin >> odpoved;

  int odkial = 1, pokial = n;   // kandidáti na poklad
  while (odkial != pokial) {
    int s = (odkial + pokial) / 2;
    int kde = 2*s - p;
    cout << "vysavaj " << kde << endl;
    cin >> odpoved;

    if (odpoved == "rovnako") {
      odkial = s; pokial = s;
      break;
    }

    bool poklad_je_mensi;
    if (kde < p) poklad_je_mensi = (odpoved == "blizsie");
    else         poklad_je_mensi = (odpoved == "dalej");

    if (poklad_je_mensi) pokial = s - 1;
    else                 odkial = s + 1;

    p = kde;
  }
  cout << "kop " << odkial << endl;
}
```

Za zmienku ešte stojí, že ak $n \leq 10^9$, tak v práve uvedenej implementácii skutočne stačia 32-bitové celé čísla. Rozmyslite si, aké najmenšie a aké najväčšie

číslo môže mať vysávané políčko.

Podúloha B: bez predlžovačky:

Minulé riešenie má tú slabinu, že políčko $2s-p$ nemusí byť medzi 1 a n. Keď nemáme predĺžovačku, môže sa stať, že nebudeme vedieť jedným ťahom zistiť, na ktorú stranu od políčka s poklad leží. Ako zabezpečiť, že sa zmestíme do limitu na počet povysávaných políčok, keď ani nevieme, koľko presne ten limit vlastne je?

Optimálny program síce nepoznáme, ale niečo o ňom vieme zistiť. Napríklad to, že v najhoršom prípade povysáva aspoň $\lfloor \log_2 n \rfloor$ políčok – každý program sa totiž dá prinútiť, aby ich povysával aspoň toľko.

Keď nejaký program testujeme, väčšinou si vopred vymyslíme, kde je poklad, a potom programu pravdivo hovoríme, ako je ďaleko. Ale tentoraz budeme zákerní a nezvolíme miesto pokladu dopredu. Budeme si pamätať, ktoré políčka sú ešte kandidáti (tie, že keby tam bol poklad, neprotirečilu by to ničomu, čo sme programu zatiaľ povedali), a vždy, keď ten program niečo povysáva, zvolíme takú odpoveď, aby tých kandidátov zostalo čo najviac. Časom síce zostane už len jeden kandidát, takže program ten poklad nájde, ale bude mu to dlho trvať. A nikto sa nemôže sťažovať, že sme podvádzali, lebo keby bol poklad naozaj na tom mieste, komunikácia Georga s programom by vyzerala úplne rovnako, a tiež by to trvalo tak dlho.

Keď si vyberáme, či programu odpovieme "bližšie", "ďalej" alebo "rovnako", možnosť "rovnako" nechceme (tam je najviac jeden kandidát, takže programu by bolo hneď jasné, kde je poklad), a z tých zvyšných dvoch vyberieme tú, kde nám zostane nadpolovičná väčšina kandidátov. Každým povysávaným políčkom zmenší program počet kandidátov najviac na polovicu, takže kým sa dostane na jediného kandidáta, musí povysávať aspoň $\lfloor \log_2 n \rfloor$ políčok.

Zistili sme, že aj optimálny program niekedy potrebuje $\lfloor \log_2 n \rfloor$ povysávaní, takže náš program ich má k dobru $2\lfloor \log_2 n \rfloor$. A akonáhle vieme, že ich môžeme spraviť až toľko, je to jednoduché: ak chceme vedieť, na ktorej strane od nejakého s poklad leží, môžme vyskúšať oboch susedov s. Tým pádom na jeden "ťah" potrebujeme dve povysávania, ale keďže ťahov nám vždy stačí $\lfloor \log_2 n \rfloor$, počet povysávaní sa zmestí do $2\lfloor \log_2 n \rfloor$.

Listing programu (C++)

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

```
int main() {
  string odpoved;
  int n;
  cin >> n;

  int odkial = 1, pokial = n;   // kandidáti na poklad
  while (odkial != pokial) {
    if (pokial - odkial == 1) {
      cout << "vysavaj " << odkial << endl;
      cin >> odpoved;
      cout << "vysavaj " << pokial << endl;
      cin >> odpoved;
      if (odpoved == "blizsie") {
        odkial = pokial;
      }
      else {
        pokial = odkial;
      }
      break;
    }

    int s = (odkial + pokial) / 2;
    cout << "vysavaj " << (s - 1) << endl;
    cin >> odpoved;
    cout << "vysavaj " << (s + 1) << endl;
    cin >> odpoved;
    if (odpoved == "blizsie") {
      odkial = s + 1;
    }
    else if (odpoved == "dalej") {
      pokial = s - 1;
    }
    else {   // "rovnako"
      odkial = s; pokial = s;
    }
  }
  cout << "kop " << odkial << endl;
}
```

B-II-4 Roboti

Podúloha A: behaj tam a späť:

Na vyriešenie tejto úlohy stačilo pomocou povolených inštrukcií prepísať program „choď doprava kým nenájdeš kamienok, choď doľava kým nenájdeš kamienok, opakuj“. Celé to môže vyzerať napríklad nasledovne:

```
1: vpravo
2: ak je kamienok, pokracuj 4
3: pokracuj 1
4: vlavo
5: ak je kamienok, pokracuj 1
6: pokracuj 4
```

Podúloha B: nájdi stred:

Existuje viacero možných riešení, my si ukážeme také, ktoré navyše po sebe aj uprace – teda vyzbiera kamienky. Budeme robiť presne to isté ako v riešení podúlohy A, len navyše vždy, keď nájdeme kamienok, ho posunieme o políčko bližšie smerom ku druhému kamienku. Keď sa oba kamienky stretnú na tom istom políčku, našli sme stred. (To spoznáme tak, že ideme položiť kamienok na políčko, kde už jeden leží. V takom prípade kamienok nepoložíme, ale naopak, zdvihneme aj ten druhý a skončíme.)

```
1: vpravo
2: ak je kamienok, pokracuj 4
3: pokracuj 1
4: zdvihni
5: vlavo
6: ak je kamienok, pokracuj 16
7: poloz
```

```
 8: vlavo
 9: ak je kamienok, pokracuj 11
10: pokracuj 8
11: zdvihni
12: vpravo
13: ak je kamienok, pokracuj 16
14: poloz
15: pokracuj 1

16: zdvihni
```

Program môžeme ešte o jednu inštrukciu skrátiť, ak si uvedomíme, že kamienky sa určite stretnú, keď budeme posúvať ľavý kamienok doprava. Pri posúvaní pravého kamienka doľava môžeme teda vynechať kontrolu.

Podúloha C: stretnutie:

Zjavne je nutné položiť nejaké kamienky. Totiž obaja roboti majú ten istý program. Ak nepoložíme nikdy žiadne kamienky, budú obaja vždy vykonávať naraz tú istú inštrukciu, a teda budú neustále robiť presne to isté – inými slovami, vzdialenosť medzi nimi sa nikdy nezmení.

Musíme teda niekedy položiť nejaký kamienok. Ak potom časom nastane situácia, v ktorej jeden robot kamienok vidí a druhý nie, môžu sa ich programy „rozsynchronizovať“ a máme šancu, aby sa niekedy neskôr aj stretli.

V našom riešení každý robot položí len jeden kamienok, a to hneď na začiatku. (Existujú však aj iné riešenia. Napríklad to naše, ako neskôr uvidíte, by rovnako dobre fungovalo, aj keby každý robot po každom kroku položil kamienok.)

Po položení kamienka sa obaja roboti vyberú doľava. Časom potom určite nastane situácia, že jeden z nich (ten, ktorý začínal viac vpravo) objaví kamienok položený druhým z nich. V tomto okamihu tento robot vie, že druhý je naľavo od neho. Čo ale s touto informáciou môže spraviť?

Hlavný trik riešenia bude v tom, že na začiatku sa obaja roboti vyberú doľava *pomaly*, len raz za niekoľko inštrukcií spravia krok. No a akonáhle jeden z robotov zistí, že on je ten viac vpravo, *zrýchli* a druhého robota dobehne.

Na spomalenie robota môžeme použiť inštrukciu `nic`, ale nie je to nutné – stačí vynechať kontrolu, či je na našom políčku kamienok. Vtedy si ale treba poriadne rozmyslieť poradie inštrukcií, aby nám nemohla vzniknúť *race condition*: ľavý robot skontroluje, že je sám, pravý príde na to isté políčko, uvidí robota, zastane, a následne sa ľavý robot, ktorý pri kontrole ešte nikoho iného nevidel, pohne doľava.

Vo vzorovom programe preto radšej použijeme navyše dve inštrukcie `nic` tak, aby prvý cyklus trval 6 sekúnd a druhý 3. Ak sa teda už pravý robot hýbe rýchlejšie, pôjde presne dvakrát tak rýchlo. Všimnite si, že tým pádom vždy, keď kontroluje ľavý robot prítomnosť pravého, kontroluje aj pravý prítomnosť ľavého.

Výsledný program (rovnaký pre oboch robotov):

```
 1: poloz

 2: vlavo
 3: nic
 4: ak je kamienok, pokracuj 8
 5: nic
 6: ak je robot, pokracuj 99
 7: pokracuj 2

 8: vlavo
 9: ak je robot, pokracuj 99
10: pokracuj 8

99: koniec
```

Riešenia celoštátneho kola kategórie A

A-III-1 Vodovody

Úsek miest, ktoré sú spojené dokopy vodovodom a majú v súčte nezápornú produkciu budeme v tomto vzorovom riešení volať *okres*. Cena okresu bude rovná súčtu cien potrubí, ktoré ho spájajú dokopy. Naším cieľom je teda rozdeliť kráľovstvo na niekoľko okresov tak, aby súčet ich cien bol najmenší možný.

Určenie produkcie úseku miest a ceny ich spojenia:

Počas hľadania optimálneho riešenia sa nám určite zíde vedieť efektívne určiť pre daný úsek miest, či majú v súčte nezápornú produkciu (t.j. či môžu tvoriť okres) a ak áno, koľko nás bude stáť ich spojenie dokopy.

Na toto sú výborným nástrojom *prefixové súčty*. Označme $\hat{p}(i)$ celkovú produkciu prvých i miest a $\hat{c}(i)$ cenu za postavenie prvých i ciest. Špeciálne teda $\hat{p}(0) = \hat{c}(0) = 0$. Prefixové súčty vieme ľahko vypočítať v lineárnom čase: napr. konkrétnu hodnotu $\hat{p}(i)$ vieme určiť ako $\hat{p}(i-1) + p_i$.

Pomocou týchto prefixových súčtov vieme potrebné údaje pre ľubovoľný úsek vypočítať v konštantnom čase: Majme úsek tvorený mestami od a po b vrátane. Celkovú produkciu tohoto úseku môžeme vyjadriť ako $\hat{p}(b) - \hat{p}(a-1)$. Ak je táto hodnota nezáporná, má zmysel sa pozrieť na to, koľko by nás stálo z daného úseku vyrobiť okres. Túto cenu (t.j. celkovú cenu za postavenie správnych $b-a$ kusov potrubia) vieme vypočítať ako $\hat{c}(b-1) - \hat{c}(a-1)$.

Riešenie v kvadratickom čase:

Všetkých možných rozdelení kráľovstva na okresy môže byť až exponenciálne veľa v závislosti od počtu miest. Dobré riešenie teda nemôžeme založiť na skúšaní všetkých možností. Presnejšie, budeme to skúšanie musieť robiť efektívnejšie.

Predstavme si, že sme sa už rozhodli, koľko miest bude tvoriť posledný okres – teda ten obsahujúci mesto n. Takéto rozhodnutie nám rozdelí kráľovstvo na dva úseky: prvý tvorený mestami 1 až x, druhý mestami $x+1$ až n, pre nejaké x. Celý druhý úsek bude jeden okres, zatiaľ čo prvý úsek môžeme ešte skúsiť ďalej deliť na viacero menších okresov.

Samozrejme, niektoré voľby x nemusia viesť k platným riešeniam. To nahliadneme ľahko – rozdeliť kráľovstvo medzi mestami x a $x+1$ môžeme len vtedy, ak má aj úsek od 1 po x, aj úsek od $x+1$ po n nezápornú celkovú

produkciu.

A akonáhle sme zvolili x, dostávame ten istý problém ako na začiatku, len už s kratšou postupnosťou: „Ako najlacnejšie vieme rozdeliť do okresov postupnosť tvorenú prvými x mestami?“

Túto otázku môžeme samozrejme zodpovedať vhodným rekurzívnym volaním nášho programu. Takto ľahko dostaneme rekurzívne riešenie s exponenciálnou časovou zložitosťou, ktoré skúša všetky možné spôsoby rozdelenia kráľovstva na okresy. Schéma takéhoto riešenia:

```
najlepsie(z):
    odpoved = cena pospajania 1..z (alebo nekonecno ak mesta 1..z netvoria okres)
    pre kazde x od 1 po z-1:
        ak mozeme 1..z rozdelit na 1..x a (x+1)..z:
            toto = najlepsie(x) + cena pospajania (x+1)..z
            odpoved = min(odpoved, toto)
    return odpoved

riesenim ulohy je najlepsie(n)
```

No a od takéhoto pomalého rekurzívneho riešenia je už len krok ku riešeniu efektívnemu. Stačí nepočítať tú istú vec zbytočne veľakrát. Presnejšie, stačí si všimnúť, že celý výpočet nášho programu spočíva v tom, že dokola voláme funkciu `najlepsie` s rôznymi parametrami – ale tento parameter nadobúda len hodnoty 1 až n. Dokola teda znova a znova zbytočne počítame tie isté hodnoty. Omnoho efektívnejšie bude použiť *memoizáciu*: akonáhle raz zistíme, že napr. `najlepsie(30)` vráti hodnotu 4700, zapamätáme si to a všetky ďalšie volania `najlepsie(30)` už prebehnú v konštantnom čase – len vrátime zapamätanú hodnotu.

Takéto riešenie má zjavne časovú zložitosť $\Theta(n^2)$: pre každé z od 1 po n sa len raz vykoná telo funkcie `najlepsie(z)`, ktorého časová zložitosť je lineárna od z.

Listing programu (C++)

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

const int NEVIEM = -1;
const long long NEKONECNO = 1LL<<60;
int N;                          // počet miest
vector<long long> P, C, sP, sC; // produkcie miest, ceny rúr, ich prefixové súčty
vector<long long> memo;         // zapamätané hodnoty funkcie najlepsie()

vector<long long> prefixove_sucty(const vector<long long> &V) {
    vector<long long> sV(1,0);
    for (long long x : V) sV.push_back( sV.back()+x );
```

```
    return sV;
}

// môže úsek miest a..b tvoriť okres?
bool moze_byt_okres(int a, int b) { return ( sP[b] - sP[a-1] ) >= 0; }
// koľko ma stojí pospájať mestá a..b?
long long cena_okresu(int a, int b) { return sC[b-1] - sC[a-1]; }

long long najlepsie(int z) {
    if (memo[z] == NEVIEM) {
        memo[z] = NEKONECNO;
        if (moze_byt_okres(1,z)) memo[z] = cena_okresu(1,z);
        for (int x=1; x<z; ++x)
            if (moze_byt_okres(1,x) && moze_byt_okres(x+1,z))
                memo[z] = min( memo[z], najlepsie(x) + cena_okresu(x+1,z) );
    }
    return memo[z];
}

int main() {
    // načítame vstup
    cin >> N;
    P.resize(N);   for (long long &p : P) cin >> p;
    C.resize(N-1); for (long long &c : C) cin >> c;
    sP = prefixove_sucty(P);
    sC = prefixove_sucty(C);
    memo.resize(N+1,NEVIEM);
    cout << najlepsie(N) << endl;
}
```

Ekvivalentné riešenie vieme zapísať aj bez rekurzie využitím *dynamického programovania*: hodnoty, ktoré si predchádzajúce riešenie počas výpočtu zapamätalo, budeme počítať v cykle s rastúcou hodnotou z.

Listing programu (C++)

```
// začiatok programu je rovnaký ako v predchádzajúcom listingu
int main() {
    cin >> N;
    P.resize(N);   for (long long &p : P) cin >> p;
    C.resize(N-1); for (long long &c : C) cin >> c;
    sP = prefixove_sucty(P);
    sC = prefixove_sucty(C);
    memo.resize(N+1,NEVIEM);
    for (int z=1; z<=N; ++z) {
        memo[z] = NEKONECNO;
        if (moze_byt_okres(1,z)) memo[z] = cena_okresu(1,z);
        for (int x=1; x<z; ++x)
            if (moze_byt_okres(1,x) && moze_byt_okres(x+1,z))
                memo[z] = min( memo[z], memo[x] + cena_okresu(x+1,z) );
    }
    cout << memo[N] << endl;
}
```

Vzorové riešenie:

Ukážeme si teraz, ako zlepšiť časovú zložitosť predchádzajúceho riešenia. Základná myšlienka zlepšenia: pri počítaní najlepšieho riešenia pre mestá 1 až z

nebudeme skúšať všetky x od 1 po $z-1$, ale použijeme vhodnú dátovú štruktúru, ktorá nám efektívne povie najlepšiu spomedzi všetkých použiteľných hodnôt x.

V prvom rade potrebujeme vedieť spomedzi všetkých x vybrať tie, ktoré zodpovedajú prípustnému rozdeleniu miest 1 až z na posledný úsek a zvyšok. Inými slovami, tie x, pre ktoré majú aj úsek 1 až x, aj úsek $x+1$ až z nezápornú celkovú produkciu. Pomocou prefixových súm môžeme tieto podmienky prepísať nasledovne: musí platiť $\hat{p}(x) \geq 0$ a zároveň $\hat{p}(z) \geq \hat{p}(x)$.

Spomedzi týchto x hľadáme to, ktoré nám dá najlepší výsledok – teda najmenší súčet už spočítaného riešenia pre úsek 1 až x a ceny za pospájanie úseku od $x+1$ po z.

Tu sa ale črtá možný problém. Nám by sa hodilo mať dátovú štruktúru, do ktorej len vkladáme nové záznamy a pýtame sa jej. Lenže hodnota, ktorú sa snažíme minimalizovať, nie je konštantná, ale závisí od aktuálnej hodnoty z. Čo s tým?

Našťastie sa obávame zbytočne, žiaden problém tu nie je. Všimnime si totiž, čo sa stane, keď prejdeme z nejakej hodnoty z na nasledujúcu, o jedno väčšiu. Pre *úplne všetky* možné výbery x cena za pospájanie posledného úseku narastie o tú istú hodnotu c_z (t.j. cenu za spojenie miest z a $z+1$). Relatívne poradie jednotlivých možností sa teda nezmení.

V našej implementácii to vyriešime nasledovne: Predstavme si, že sme práve spočítali, že najlacnejší spôsob, ako pospájať prvých z miest, má cenu φ. Do našej dátovej štruktúry si teraz uložíme pre index z cenu $\varphi + \hat{c}(n-1) - \hat{c}(z)$. Slovne: k cene za optimálne riešenie pre mestá 1 až z pripočítame cenu za spojenie všetkých ostatných miest ($z+1$ až n) dokopy. Z uloženej ceny vieme ľahko v konštantnom čase určiť cenu pre ľubovoľný iný, kratší úsek pospájaných miest na konci.

V pseudokóde si teda naše nové riešenie môžeme zapísať nasledovne:

```
zober prazdnu datovu strukturu D

pre kazde z od 1 po n:
    odpoved = cena pospajania 1..z (alebo nekonecno ak mesta 1..z netvoria okres)
    v D najdi pripustny zaznam s najmensou cenou
    ak si taky zaznam nasiel:
        vypocitaj z jeho ceny cenu len pre mesta 1..z
        uloz ju do odpoved[z]
    vloz do D zaznam o odpoved[z]
```

Záznamy v našej dátovej štruktúre budú mať dve zložky: prefixovú sumu $\hat{p}(z)$, aby sme vedeli, kedy danú hodnotu môžeme neskôr použiť, a hodnotu $odpoved[z] + \hat{c}(n-1) - \hat{c}(z)$, ktorú sa snažime minimalizovať.

Existuje viacero možných implementácií, napr. pomocou intervalového stromu: usporiadame si všetky hodnoty prefixových súm $\hat{p}$ a následne nad nimi postavíme intervalový strom, do ktorého budeme vkladať upravené vypočítané odpovede.

My sme si zvolili implementáciu pomocou množiny – vyvažovaného stromu, v ktorom si pamätáme vyššie popísané záznamy ako usporiadané dvojice – ale pamätáme si len tie, ktoré majú teoretickú šancu niekedy byť použité. Akonáhle teda vkladáme záznam (a, b), zmažeme všetky záznamy (c, d) pre ktoré súčasne platí $c \geq a$ a $d \geq b$ – totiž kedykoľvek, kedy by sme mohli použiť záznam (c, d), môžeme použiť aj záznam (a, b) a ten je lepší. Takto dosiahneme, že postupnosť dvojíc uložených v našej množine je *súčasne* usporiadaná vzostupne podľa prefixovej sumy aj zostupne podľa odpovede.

Ak teda napr. máme v nejakom okamihu v našej dátovej štruktúre uložené dvojice $(2, 10)$ a $(4, 7)$, môžeme to chápať nasledovne: kedykoľvek, keď v budúcnosti budeme spracúvať nejaké z, pre ktoré $\hat{p}(z) \geq 2$, existuje k nemu x, pomocou ktorého optimálne riešenie pre aktuálne z vypočítame z čísla 10 odpočítaním vhodnej konštanty. A ak dokonca platí $\hat{p}(z) \geq 4$, tak vieme mať ešte o 3 lepšie riešenie pre aktuálne z.

Aká je časová zložitosť tohto riešenia? Do usporiadanej množiny vložíme nanajvýš n záznamov a každý z nich nanajvýš raz zmažeme. Navyše nanajvýš n-krát budeme v našej množine vyhľadávať najlepšiu použiteľnú dvojicu. Každú z týchto operácií vieme spraviť v čase $O(\log n)$, celková časová zložitosť je teda $O(n \log n)$.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <set>
using namespace std;

const long long NEKONECNO = 1LL<<60;
int N;                          // počet miest
vector<long long> P, C, sP, sC; // produkcie miest, ceny rúr, ich prefixové súčty
vector<long long> odpoved;      // odpoved[z] je optimálne riešenie pre mestá 1..z

vector<long long> prefixove_sucty(const vector<long long> &V) {
    vector<long long> sV(1,0);
    for (long long x : V) sV.push_back( sV.back()+x );
    return sV;
}

// môže úsek miest a..b tvoriť okres?
bool moze_byt_okres(int a, int b) { return ( sP[b] - sP[a-1] ) >= 0; }
// koľko ma stojí pospájať mestá a..b?
long long cena_okresu(int a, int b) { return sC[b-1] - sC[a-1]; }
```

```
typedef pair<long long, long long> zaznam;
set<zaznam> D;                  // dátová štruktúra, v ktorej máme naše záznamy

void vloz(long long a, long long b) {
    // overíme, či nemáme lepšiu alebo rovnú možnosť ako (a,b)
    auto it = D.lower_bound(zaznam(a+1,0));
    --it;
    if (it->second <= b) return;
    // vyhádžeme možnosti od ktorých je (a,b) lepšie
    ++it;
    while (it != D.end() && it->second >= b) { auto jt=it; ++it; D.erase(jt); }
    // pridáme (a,b)
    D.insert(zaznam(a,b));
}

int main() {
    cin >> N;
    P.resize(N);   for (long long &p : P) cin >> p;
    C.resize(N-1); for (long long &c : C) cin >> c;
    sP = prefixove_sucty(P);
    sC = prefixove_sucty(C);
    odpoved.resize(N+1,NEKONECNO);
    D.insert(zaznam(0,NEKONECNO)); // zarážka
    for (int z=1; z<=N; ++z) if (moze_byt_okres(1,z)) {
        odpoved[z] = cena_okresu(1,z);

        // nájdi v D lepšiu možnosť
        auto it = D.lower_bound(zaznam(sP[z]+1,0));
        --it;
        if (it->second != NEKONECNO) odpoved[z] = it->second - cena_okresu(z,N);

        // vlož do D práve spočítanú odpoveď
        vloz( sP[z], odpoved[z] + cena_okresu(z+1,N) );
    }
    cout << odpoved[N] << endl;
}
```

A-III-2 Dievka na vydaj

Malé hodnoty k:

Pozrime sa najskôr na riešenia, ktoré využívajú fakt, že k je relatívne malé (do 10^5). Pre $n = 2$ vieme použiť prístup ako v mergesorte. Na začiatku mám dva ukazovatele, oba nastavené na začiatok k-tic. Následne zistím, ktorý z prvkov, na ktoré ukazujú, je väčší a to bude najbohatší pytač. Daný ukazovateľ posuniem o jedna dozadu. Tento postup zopakujem dokopy k-krát a posledné vybraté číslo bude pytač, ktorého hľadáme. Takto dosiahneme časovú zložitosť $O(k)$.

Ako to zovšeobecníme pre $n > 2$? Vtedy si musíme udržiavať n ukazovateľov a v každej iterácii zistiť maximum zo všetkých prvkov na ktoré ukazujú. Dôležité je uvedomiť si, že aj keď nepoznáme presné hodnoty prvkov, na ktoré ukazujeme, vieme ich porovnávať. To znamená, že vieme upraviť všetky algoritmy, ktoré

fungujú na princípe porovnávania tak, aby fungovali aj pre náš problém. V tomto prípade sa nám oplatí použiť haldu. Nad našimi n ukazovateľmi si teda postavíme maximovú haldu, z ktorej v každej iterácii vyberieme najväčší prvok, daný ukazovateľ posunieme o jedno dozadu a tento nový prvok vložíme spať do haldy. To nám zabezpečí časovú zložitosť $O(k \log n)$.

Veľké hodnoty k:

Zo zadania však vyplýva, že by bolo dobré vedieť riešiť túto úlohu aj pre veľké hodnoty k, preto nechceme, aby časová zložitosť nášho programu bola lineárna od k. Opäť začnime prípadom, keď $n = 2$. Povedzme si, že k-ty najväčší prvok sa nachádza v prvej skupine. A to, ktorý konkrétny prvok to je, budeme binárne vyhľadávať. Povieme si, že x-tý najväčší prvok v prvej skupine bude výsledok. Ako to overíme? Máme dve možnosti, buď v druhej skupine binárne dohľadáme, ktorý prvok je najmenší väčší ako prvok x, a overíme, či je to prvok $k-x$, alebo sa rovno spýtame, či $(k-x)$-tý prvok je najmenší väčší ako x a v konštantnom čase overíme. To nám dá zložitosť $O(\log^2 k)$, respektíve $O(\log k)$.

Skúsme teraz zovšeobecniť toto riešenie aj pre n skupín. Postupne vyskúšame n možností pre to, v ktorej skupine leží výsledok. Následne budeme výsledok binárne vyhľadávať a overovať to tak, že vo zvyšných skupinách binárne vyhľadám prvok, ktorý je najmenší väčší ako ten, čo sme si zvolili. Z toho spočítam celkový počet prvkov väčších alebo rovných práve skúšanému a to porovnám so želanou hodnotou k. Toto nám zaručí časovú zložitosť $O(n^2 \log^2 k)$.

Ako si pamätať, čo sme už zistili:

Predchádzajúce riešenie má zjavnú slabinu: pre každú skupinu začíname hľadanie odznova a nevyužívame pri tom, čo sme sa už skôr dozvedeli. Ako to ale využiť vieme?

Budeme si udržiavať množinu *kandidátov* – princov, ktorí ešte môžu byť k-tym najbohatším zo všetkých. Na začiatku množinu kandidátov zjavne tvorí prvých k princov z každej skupiny. Ako sa budeme dozvedať nové informácie, bude sa množina kandidátov zmenšovať, až v nej nakoniec ostane len jediný princ. Navyše, ak o nejakom princovi zistíme, že medzi kandidátov nepatrí, lebo je príliš chudobný, nebudú medzi kandidátov patriť ani princovia z jeho skupiny, ktorí sú za ním v poradí – tí sú ešte chudobnejší. A podobne, ak je nejaký princ príliš bohatý, môžeme spolu s ním vylúčiť všetkých z jeho skupiny, ktorí sú od neho bohatší. Množina kandidátov sa nám teda bude pamätať ľahko: v každej zo skupín budú kandidáti vždy tvoriť súvislý úsek.

Zmenšovať množinu kandidátov vieme napríklad postupom, ktorý sme si už

načrtli vyššie, ale pre poriadok ho popíšeme znova a poriadne: Vyberieme si nejakého princa p. Zadarmo vieme, koľko princov v jeho skupine je od neho bohatších. Pre každú zo zvyšných $n-1$ skupín môžeme použiť binárne vyhľadávanie a v čase $O(\log k)$ určiť, koľko princov v danej skupine je od princa p bohatších. Takže v celkovom čase $O(n \log k)$ vieme zistiť presné poradie princa p.

No a v tomto okamihu nastávajú tri možné prípady. Ak máme šťastie a princ p je presne k-ty, tak sme vyriešili úlohu a končíme. Ak je princ p chudobnejší ako k-ty, môžeme princa p a všetkých chudobnejších od neho vylúčiť spomedzi kandidátov. (Vďaka binárnym vyhľadávaniam, ktoré sme práve dokončili, vieme v každom riadku, ktorých princov ideme vylúčiť.) No a ak je princ p príliš bohatý, vylúčime spomedzi kandidátov okrem p aj všetkých princov bohatších od neho.

Teraz by sa zdalo, že ľahko vylepšíme predchádzajúce riešenie. V okamihu, keď ideme spracovať i-tu skupinu, jej aktuálny úsek kandidátov môže byť zmenšený o to, čo sme sa už dozvedeli pri spracúvaní predchádzajúcich skupín. Môže sa teda stať, že budeme potrebovať menej otázok pri binárnom vyhľadávaní správneho princa v i-tej skupine. (Uvedomte si, že za princa p má vždy zmysel voliť len jedného z aktuálnych kandidátov, inak sa nedozvieme nič nové.)

V mnohých situáciách by toto zlepšenie skutočne pomohlo. Problémom je však najhorší možný prípad. Ten aj po našom vylepšení zostáva stále rovnaký. Predstavte si napríklad vstup, kde pre každé i platí, že všetci pytači v i-tej skupine sú chudobnejší od pytačov v nasledujúcich skupinách. Stále teda máme riešenie s časovou zložitosťou v najhoršom prípade až $\Theta(n^2 \log^2 k)$.

Využitie náhody:

Ocitli sme sa práve vo veľmi podobnej situácii ako napr. pri triedení QuickSort: hoci skoro každá voľba pivota (v našom prípade princa p) je dobrá a teda výrazne nám zmenší množinu kandidátov, existujú akési špecifické zlé vstupy, kedy ako na potvoru robíme samé zlé voľby a teda výpočet beží dlho.

No a podobne ako u QuickSortu, aj tu nám pomôže náhoda. Dostaneme tak riešenie, ktoré *nebude mať žiadne zlé vstupy*. Pre *úplne hocijaký* vstup bude platiť, že čas behu programu závisí len od toho, ako dobre sa nám bude dariť pri náhodnom výbere pivotov. No a bude platiť, že *v priemernom (očakávanom) prípade* sa nám bude dariť dostatočne dobre. Pozrime sa teda na to poriadnejšie.

Nový, vylepšený algoritmus: Na začiatku inicializujeme množinu kandidátov na prvých k princov z každej skupiny. No a kým máme viac ako jedného kandidáta, tak dokola opakujeme: za princa p zvolíme *náhodného* spomedzi všetkých kandidátov, v čase $O(n \log k)$ zistíme poradie princa p, a podľa neho upravíme

množinu kandidátov.

Odhad časovej zložitosti len načrtneme. Uvažujme ľubovoľnú situáciu počas behu algoritmu. Nech má aktuálna množina kandidátov veľkosť c. Potom ľahko spočítame, že v nasledujúcom kole zahodíme v priemere aspoň $c/4$ kandidátov. No a keďže $(3/4)^3 < 1/2$, môžeme očakávať, že za každé tri kolá sa nám počet kandidátov v priemere zmenší aspoň na polovicu. V priemernom prípade bude teda kôl nanajvýš $3 \log_2(nk)$, čo vďaka nerovnosti $n < k$ môžeme odhadnúť ako $O(\log k)$ kôl.

V rámci každého kola potrebujeme najskôr vybrať náhodného kandidáta (rozmyslite si, že to vieme ľahko spraviť napr. v čase $\Theta(n)$) a potom ho použiť ako pivota (na čo treba čas $O(n \log k)$). Dokopy teda dostávame riešenie, ktorého očakávaná časová zložitosť (na ľubovoľnom vstupe) je $O(n(\log k)^2)$.

(Poznamenáme ešte, že sa dá dokázať aj silnejšie tvrdenie, ktoré hovorí, že pre dostatočne veľké n a k je pravdepodobnosť toho, že by náš algoritmus bežal rádovo dlhšie, zanedbateľne malá.)

Vzorové riešenie:

Vráťme sa teraz späť k prvému riešeniu, kde sme používali haldu. Problém bol, že sme zo skupín vždy vyberali len po jednom prvku, takže to trvalo príliš dlho. Skúsme teda odhadzovať viac prvkov naraz. Rozďeľme si každú skupinu princov na bloky veľkosti s. (To, aké veľké s bude, sa rozhodneme neskôr.) Bloky budeme porovnávať tak, že porovnáme princov, ktorými začínajú. Zoberieme maximovú haldu do ktorej vložíme n záznamov: prvý blok z každej skupiny princov. Následne z haldy postupne vyberieme a zahodíme $\lfloor k/s \rfloor$ blokov (pričom zakaždým, keď nejaký blok vyhodíme, vložíme namiesto neho nasledujúci blok z tej istej skupiny).

Nech p je prvý (teda najväčší) prvok v poslednom vybranom bloku. Všetky prvky, ktoré sme nevybrali, sú nutne menšie ako p – lebo všetky ostatné bloky v halde začínajú prvkami menšími ako p a všetky ostatné prvky sú od týchto začiatočných prvkov menšie. No a vybratých prvkov bolo nanajvýš k, preto platí, že princ, ktorého hľadáme, je buď p, alebo je od neho chudobnejší.

Čo ešte vieme o hľadanom princovi povedať? Predstavme si, že sme pri práve popísanom procese z jednej skupiny postupne vyhodili bloky b_1 až b_t. Posledný blok b_t začína princom aspoň tak bohatým ako p. (Je to presne p, ak je b_t úplne posledný vyhodený blok, inak je to iný, bohatší princ.) To ale znamená, že všetky bloky, ktoré sme z danej skupiny vyhodili skôr, obsahujú samých princov bohatších ako p – a teda bohatších ako ten, ktorého hľadáme. Všetkých týchto princov môžeme teda poslať domov a príslušne zmenšiť poradové číslo k

princa, ktorého hľadáme.

Aby to bolo rozumne rýchle, s si na začiatku zvolíme ako najbližšiu mocninu dvoch menšiu ako k. Následne budeme celý vyššie popísaný proces viackrát opakovať, pričom zakaždým zmenšíme s na polovicu. V poslednej iterácii bude $s = 1$, čím dostaneme algoritmus popísaný v časti o malej hodnote k.

Odhadnime teraz časovú zložitosť tohto algoritmu. Zjavne prebehne $\log k$ iterácií vyššie popísaného procesu. V prvej iterácii (vďaka začiatočnej voľbe s) budú výbery z haldy nanajvýš dva.

Všimnime si teraz ľubovoľnú iteráciu i. Máme nejakú aktuálnu hodnotu k (poradové číslo princa, ktorého práve hľadáme) a s. Postupne vyberieme $\lfloor k/s \rfloor$ blokov z haldy a následne z každej skupiny všetky vybraté bloky okrem posledného pošleme domov. Domov sme teda poslali aspoň $\lfloor k/s \rfloor - n$ blokov princov. Inými slovami, spomedzi k najbohatších princov nám ich ostalo určite menej ako $s(n+1)$, a teda pre nové poradové číslo k' princa, ktorého hľadáme v nasledujúcej iterácii, bude platiť $k' < s(n+1)$.

To ale znamená, že v iterácii $i+1$, keď budeme mať bloky veľkosti $s/2$, vyberieme z haldy dokopy $\lfloor k'/(s/2) \rfloor \leq 2(n+1)$ blokov.

No a vyššie popísaná úvaha platí pre každé i. V každej iterácii bude teda najviac $2(n+1)$ výberov z haldy. No a každý z nich trvá $O(\log n)$, čím dostávame celkovú časovú zložitosť $O(n \log n \log k)$.

Poznámka na záver: Existuje aj komplikovanejšie riešenie s časovou zložitosťou $O(n \log k)$.

Listing programu (C++)

```
#include <vector>
#include <queue>
#include <cstdio>
using namespace std;

struct princ {
    int s; //číslo skupiny
    int i; //pozícia v skupine
};

bool operator<(const princ &p, const princ &d) {
    return bohatsi(d.s+1,d.i+1,p.s+1,p.i+1)<0;
}

int main() {
    int n,k;
    scanf("%d %d ",&n,&k);
    vector<int> pocty(n);
    for(int i=0; i<n; i++) {
```

```
        scanf("%d ",&pocty[i]);
    }
    //najbližšia mocnina 2
    int s=1;
    while(2*s <= k) s*=2;
    //halda na skupiny princov
    priority_queue<princ> halda;
    vector<int> pozicie(n,0);
    while(s>=1) {
        halda=priority_queue<princ>();
        for(int i=0; i<pozicie.size(); i++) {
            //vráťme poslednú skupinu
            if(pozicie[i] > 0){
                pozicie[i]-=2*s;
                k+=2*s;
            }
            //naplňme haldu
            princ p={i,pozicie[i]};
            halda.push(p);
        }
        //skúšame kroky veľkosti s
        for(; k-s>0; k-=s) {
            princ p = halda.top();
            halda.pop();
            p.i=pozicie[p.s]+=s;
            if(p.i < pocty[p.s]) halda.push(p);
        }
        s/=2;
    }
    //hľadaný princ je teraz navrchu haldy
    princ hladany = halda.top();
    printf("%d %d\n",hladany.s+1,hladany.i+1);
    return 0;
}
```

A-III-3 Log-space výpočty

Podúloha A:

Na bežnom počítači by sme túto úlohu vyriešili prehľadávaním. O každom prvku by sme si pamätali, či už bol navštívený. Pri počítaní cyklov postupne prechádzame prvkami permutácie a vždy, keď nájdeme nenavštívený prvok, zvýšime si počet cyklov, prejdeme celý cyklus obsahujúci daný prvok a označíme všetky jeho prvky ako navštívené.

Na takéto riešenie by sme však potrebovali pomocnú pamäť lineárnu od dĺžky permutácie, aby sme pre každý prvok vedeli, či sme ho navštívili alebo nie. Log-space program si však nemôže pamätať niečo pre každý prvok, dokonca ani pre každý cyklus nie – môže ich byť až $O(n)$. (Napr. pre $n = 8$ má permutácia 12345678 presne 8 cyklov, permutácia 21436587 ich má $n/2 = 4$.)

Ako na to teda s logaritmickou pamäťou?

Jedna sľubne vyzerajúca cesta je priradiť prvkom rôzne váhy: keď spracúvam prvok, zistím si dĺžku d cyklu, na ktorom leží, a k výsledku pripočítam hodnotu $1/d$. Keď teda mám napríklad konkrétny cyklus dĺžky 3, pre každý jeho prvok pripočítam k výsledku hodnotu 1/3, čím dostanem v súčte 1. Toto riešenie síce funguje, ale samo o sebe nie je log-space, keďže naše programy nevedia pracovať s reálnymi číslami.

Mohli by sme sa pokúsiť reprezentovať priebežný súčet ako zlomok: v jednej premennej si budeme pamätať jeho čitateľ, v druhej menovateľ. Takéto riešenie však opäť nie je log-space. Prečo? Pretože čitateľ aj menovateľ môžu byť veľmi veľké v porovnaní s n. Uvedomte si totiž, že vo všeobecnosti môže byť niekedy uprostred výpočtu menovateľ pamätaného zlomku najmenším spoločným násobkom dĺžok všetkých cyklov danej permutácie.

(Pomocou vysokoškolskej matematiky sa dá dokázať, že na uloženie menovateľa treba $\Theta(\sqrt{n \log n})$ bitov, čo je priveľa. Jednoduchší príklad toho, že $O(\log n)$ bitov nestačí: predstavte si permutáciu, ktorá má zhruba $\sqrt{n}$ cyklov, každý inej dĺžky, pričom tie dĺžky sú všetky približne rovné $\sqrt{n}$. Aký veľký môže byť približne menovateľ zlomku počas spracúvania takejto permutácie? Koľko bitov treba na jeho uloženie?)

Je možné, že sa toto riešenie dá upraviť do funkčnej podoby – a to využitím pozorovania, že súčet je vždy celé číslo, a teda nám neprekážajú rozumne malé zaokrúhľovacie chyby. Takáto úprava by zrejme bola veľmi komplikovaná, nehovoriac o dôkaze jej správnosti. Existuje však aj omnoho jednoduchšie riešenie.

Namiesto toho, aby sme pre každý z d prvkov na cykle zarátali k výsledku $1/d$, jednoducho pri jednom z nich zarátame 1 a pri ostatných 0. Ako toto ale zabezpečiť? Potrebujeme si pre každý cyklus určiť jeho jedného *reprezentanta*. Dobrou voľbou je napríklad najmenší prvok ležiaci na danom cykle.

Celé riešenie si teda ľahko zhrnieme jednou vetou: Pre každý prvok permutácie prejdeme celý jeho cyklus, a ak na ňom nestretneme žiaden menší prvok, tak zväčšíme o 1 počet nájdených cyklov.

Na toto nám s prehľadom stačí konštantne veľa premenných, ako demonštruje aj náš program.

Listing programu (Pascal)

```
var n: integer;                  { vstup: dĺžka permutácie }
    P : array [1..n] of integer; { vstup: permutácia }
    c : integer;                 { výstup: počet cyklov}
    poc, min : integer;          { počet cyklov, min. videný prvok }
    v, i : integer;              { v ktorom vrchole som, z ktorého som začínal }
```

```
begin
    poc := 0
    for i := 1 to n do begin
      min := i
      v := i
      repeat
        v := P[v];
        if min > v then min := v;
      until v = i;
      if min = i then inc(poc);
    end;
    c := poc;
end.
```

Iné riešenie. Nápad „pre každý prvok ležiaci na cykle dĺžky d k výsledku pripočítam hodnotu $1/d$“ sa dá prerobiť na funkčné riešenie nasledovne: Postupne skúšame všetky d od 1 po n. Pre konkrétne d postupne prejdeme všetky prvky a spočítame, koľko z nich leží na cykle dĺžky presne d. Takto získaný počet vydelíme d (čo vieme spraviť v celých číslach) a pripočítame k výsledku.

Ešte iné riešenie. Postupne pre každé x si položíme otázku „leží x na niektorom z cyklov obsahujúcich prvky 1 až $x-1$?“ Otázku zodpovieme tak, že pre každé y od 1 po $x-1$ prejdeme cyklus obsahujúci y. Ak nikdy x nestretneme, zväčšíme si počet nájdených cyklov.

Podúloha B:

S luxusnou lineárnou pamäťou by nebol problém naprogramovať prehľadávanie stromu napríklad do hĺbky. Pamätáte si krajské kolo? Tam sme spolu prišli na to, ako aj v log-space vieme spraviť prehľadávanie stromu. Okrem prvku, na ktorom práve sme, si pamätáme aj prvok, z ktorého sme sem prišli. Keď chceme ísť ďalej, ideme do suseda s najbližším väčším číslom. Dokázali sme si, že takéto prehľadávanie naozaj dokola chodí po celom strome.

Do tohoto prehľadávania však nestačí len priamočiaro pridať počítanie spravených krokov. Totiž kým sa zo začiatočného vrcholu u dostaneme do v, môže sa stať, že nepôjdeme priamo, ale niektoré vrcholy navštívime viackrát.

Predstavme si, že začneme náš strom prehľadávať z vrcholu u a časom prídeme do v. Čo vieme teraz povedať o hľadanej ceste? Nevieme síce, ako vyzerá celá, ale vieme, že jej posledná hrana je tá, ktorou sme práve prišli do v.

Prečo? Zakoreňme si strom vo vrchole v. Vrchol u leží v niektorom z podstromov. Nech e je hrana, ktorá vedie z v do daného podstromu. Do iných podstromov sa nemáme ako dostať bez toho, aby sme šli cez hranu e. Ale akonáhle ňou prejdeme, sme vo v a prehľadávanie končí.

Zistili sme teda takto poslednú hranu želanej cesty. Nech je to (x, v) pre

nejaký vrchol x. Zvýšime si počet hrán na ceste, posunieme si cieľ v do vrcholu x a ak ešte neplatí $u = v$, začneme celý postup odznova.

Ku prehľadávaniu z krajského kola sme pridali len konštantný počet premenných, stále teda máme log-space program. Za povšimnutie ešte stojí, že najkratšia cesta má dĺžku menej ako n, celý postup teda budeme opakovať menej ako n-krát.

Listing programu (Pascal)

```
var n, u, v : integer;              { vstup: počet vrcholov, štart a cieľ }
    A, B : array [1..m] of integer; { vstup: hrany grafu }
    d : integer;                    { výstup: dĺžka najkratšej cesty }
    start, ciel, v, z, tmp, hran : integer;

function dalsia_hrana (w, z : integer) : integer;
var i, minn, nextn : integer;
begin
    minn := z; { sused s najmenším číslom }
    nextn := m+1; { sused s najmenším číslom väčším ako z }
    for i := 1 to m do begin
        if A[i] = w then begin
            if B[i] < minn then minn := B[i];
            if (B[i] > z) and (B[i] < nextn) then nextn := B[i];
        end;
        if B[i] = w then begin
            if A[i] < minn then minn := A[i];
            if (A[i] > z) and (A[i] < nextn) then nextn := A[i];
        end;
    end;
    { ak neexistuje sused s väčším číslom, vrátime najmenšieho suseda }
    if nextn <= m then dalsia_hrana := nextn
    else dalsia_hrana := minn;
end;

begin
    start := u;
    ciel := v;
    hran := 0;
    repeat
      v := start
      z := -1;
      repeat
          tmp := dalsia_hrana (v, z);
          z := v;
          v := tmp;
      until v = ciel;
      inc(hran);
      ciel = z;
    until start = ciel;
    d := hran;
end.
```

Iné funkčné riešenie zistí dĺžku cestu tak že určí počet vrcholov na nej – pre každý vrchol x overíme, či sa prehľadávanie z u dostane do vrcholu v, ak ho nepustíme cez x. Podobnou možnosťou je overiť to isté postupne pre každú hranu.

A-III-4 Špiónske voľby

Preložme si najprv úlohu do teórie grafov. Vrcholy grafu budú predstavovať špiónov a hrany budú spájať špiónov, ktorí sa poznajú.

Pomalé riešenia:

Existujú dve pomerne priamočiare pomalé riešenia. Prvé z nich si okrem siete špiónov pamätá názor každého špióna. Keď treba zmeniť špiónov názor, tak to dokáže v konštantnom čase. Na otázku koľko bodov získali kandidáti odpovedá tak, že prejde celú sieť a pre každú dvojicu špiónov, ktorý sa poznajú zistí ako hlasovali a následne pripočíta patričné body. Toto celé beží ale v lineárnom čase.

Druhé riešenie si navyše pamätá aktuálny stav bodov. Takže na otázku o bodoch vie odpovedať v konštatnom čase. Problém ale nastáva so zmenením názoru špióna. Keď špión x zmení názor, tak sa treba pozrieť na každého jeho suseda a na základe toho prepočítať body. Toto môže byť síce v niektorých prípadoch rýchle (lebo v priemere špión pozná najviac 6 iných špiónov), ale stále môžu nastať extrémne pomalé prípady (predstavte si sieť, kde špión 1 je spojený s každým a nik iný sa nepozná a následne špión 1 veľmi často mení názor). Takže v najhoršom prípade nás zmena názoru bude stáť lineárny čas od veľkosti siete.

Stredne dobré riešenie:

Vrcholy si môžeme rozdeliť na *malé* (stupňa do 6) a *veľké*. V zadaní bolo pre niektoré testovacie sady zaručené, že žiadne dva veľké vrcholy nesusedia.

Ako vieme riešiť takéto vstupy? Pre každý veľký vrchol si budeme pamätať tabuľku s informáciou, ktorého kandidáta volí koľko jeho susedov. Ak zmení názor malý vrchol, prejdeme jeho susedov, a tým, ktorí sú veľkí, upravíme tabuľku. Ak zmení názor veľký vrchol, použijeme v ňom zapamätanú tabuľku na to, aby sme dopočítali výsledok.

Takéto riešenie má konštantnú časovú zložitosť na operáciu pre sady, v ktorých žiadne dva veľké vrcholy nesusedia. (Jeho vhodným zovšeobecnením vieme dostať riešenie, ktoré bude mať vždy časovú zložitosť $O(\sqrt{n})$ na operáciu.)

Užitočné vlastnosti siete špiónov:

V tomto vzorovom riešení a aj v domácom kole sme spomínali, že priemerný počet susedov vrchola je maximálne 6. Z toho vyplýva, že v grafe vždy existuje vrchol, ktorý má nanajvýš 6 susedov. Keď tento vrchol z grafu odstránime, tak v grafe, ktorý nám ostal, zase existuje vrchol, ktorý má najviac 6 susedov.

Takto vieme postupne z grafu odstrániť všetky vrcholy. Navyše si môžeme zapamätať poradie, v akom sme vrcholy odstránili: $p_1, p_2, \ldots, p_n$. V tomto po-

radí nám platí, že každý vrchol p_i má maximálne 6 hrán do vrcholov $p_{i+1}, \ldots, p_n$. Tieto hrany si osobitne zapamätáme a nazveme ich *dopredné hrany*. Ale pozor, stále môže mať vrchol p_i veľmi veľa susedov medzi vrcholmi $p_1, \ldots, p_{i-1}$. Týchto nazveme *zadní susedia*.

Rozmyslite si, že toto predpočítavanie vieme spraviť v lineárnom čase od počtu vrcholov. Stačí si pre každý vrchol pamätať koľko má ešte neodstránených susedov a keď toto číslo klesne na 6, tak ho zaradiť do fronty.

Vzorové riešenie:

Pre každý vrchol si budeme pamätať, ako momentálne hlasuje. Navyše si budeme pre každý vrchol p_i a pre každého kandidáta j pamätať hodnotu $q_{i,j}$: počet zadných susedov vrcholu p_i, ktorí volia kandidáta j.

Nech teraz zmení názor vrchol p_i. V konštatnom čase prejdeme dopredné hrany z neho a pozrieme sa, koho volia dotyční (nanajvýš šiesti) susedia. Následne v konštantnom čase prejdeme tabuľku zapamätanú vo vrchole p_i, čím naraz spracujeme všetkých jeho zadných susedov.

Hotovo? Ešte nie. Síce sme už spočítali nové počty bodov kandidátov, potrebujeme ešte ale upraviť niektoré hodnoty v tabuľke q. Zmenil sa totiž práve hlas vrcholu p_i. Čo treba zmeniť? Tabuľky pre tie vrcholy, do ktorých vedú dopredné hrany z vrcholu p_i. No a takých je najviac 6, v každej meníme dve hodnoty, dokopy teda v tabuľke q spravíme najviac 12 zmien. S využitím dopredných hrán vieme tieto zmeny ľahko spraviť v konštantom čase.

Celkovo teda vieme v konštantom čase zmeniť názor špióna a v konštantom čase povedať body jednotlivých kandidátov. Navyše potrebujeme spraviť predpočítanie v čase $O(n)$. Na uloženie grafu a tabuliek nám stačí $O(n)$ pamäte.

Listing programu (C++)

```
#include <vector>
#include <queue>
#include <cstdio>
using namespace std;

static vector<vector<int> > g;  // cely graf
static vector<vector<int> > gp; // dopredne hrany
static vector<vector<int> > q;
static vector<int> votes;
static vector<int> state;

void spioni (int n, int poznaju_sa[][2], int voli[]) {
  g.resize(n);
  gp.resize(n);
  state.resize(5);
  votes.resize(n);
  q.resize(n, vector<int>(5));
```

```
  // pocty nevymazanych susedov
  vector<int> ps(n);
  for (int i = 0; i < m; i++) {
    g[poznaju_sa[i][0]].push_back(poznaju_sa[i][1]);
    g[poznaju_sa[i][1]].push_back(poznaju_sa[i][0]);
    ps[poznaju_sa[i][0]]++;
    ps[poznaju_sa[i][1]]++;
    if (voli[poznaju_sa[i][0]] == voli[poznaju_sa[i][1]]) {
      state[voli[poznaju_sa[i][0]]]++;
    }
  }

  queue<int> fr;
  // oznacenie pre vymazane vrcholy
  vector<bool> vymaz(n, false);
  vector<bool> sprac(n, false);
  for (int i = 0; i < n; i++) {
    if (ps[i] <= 6) {
      fr.push(i);
      vymaz[i] = true;
    }
  }

  while (!fr.empty()) {
    int x = fr.front(); fr.pop();
    sprac[x] = true;
    for (int i = 0; i < g[x].size(); i++) {
      if (sprac[g[x][i]]) continue;
      gp[x].push_back(g[x][i]);
      if (vymaz[g[x][i]]) continue;

      ps[g[x][i]]--;
      if (ps[g[x][i]] <= 6) {
        fr.push(g[x][i]);
        vymaz[g[x][i]] = true;
      }
    }
  }

  for (int i = 0; i < n; i++) {
    votes[i] = voli[i];
    for (int j = 0; j < gp[i].size(); j++) {
      q[gp[i][j]][votes[i]]++;
    }
  }
}

void zmen_nazor (int spion, int voli) {
  // najprv odcitame co treba
  state[votes[spion]] -= q[spion][votes[spion]];
  for (int i = 0; i < gp[spion].size(); i++) {
    if (votes[spion] == votes[gp[spion][i]]) {
      state[votes[spion]]--;
    }
    q[gp[spion][i]][votes[spion]]--;
  }

  votes[spion] = voli;
  state[votes[spion]] += q[spion][votes[spion]];
  for (int i = 0; i < gp[spion].size(); i++) {
    if (votes[spion] == votes[gp[spion][i]]) {
      state[votes[spion]]++;
    }
    q[gp[spion][i]][votes[spion]]++;
```

```
  }
}

void pocet_bodov (int pocet[5]) {
  for (int i = 0; i < 5; i++) {
    pocet[i] = state[i];
  }
}
```

A-III-5 Uhlopriečky

Úloha môže na prvý pohľad pôsobiť trocha odstrašujúco. Hráme sa s nejakým n-uholníkom plným uhlopriečok. Tie navyše spĺňajú nejaké čudesné pravidlo, že každý z $n-2$ trojuholníkov má aspoň jednu hranu spoločnú s pôvodným n-uholníkom.

Preto prvá vec, o čo chceme robiť, je uvidieť zadanie z iného uhla, nájsť v ňom niečo jednoduché, čo autor tak usilovne schovával.

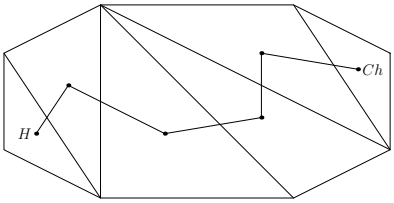
Ohmatanie úlohy:

Najprv si trocha preformulujme cieľ hry. Podľa zadania vyhrá hráč, ktorý spraví červený trojuholník. O ťah skôr to však znamená, že prehrá hráč, ktorý ofarbí druhú hranu nejakého trojuholníka. Trojuholníky, ktoré už majú nejakú hranu zafarbenú, nazvime obsadené. Strany obsadených trojuholníkov nazvime obsadené hrany.

Zjavne teda dobrá stratégia je neofarbovať obsadené hrany, kým sa to dá. Hráč, ktorý už nemá na výber neobsadené hrany, prehrá.

Teraz poďme skúmať to divné pravidlo o povolených trojuholníkoch. Čo znamená, že má trojuholník aspoň jednu hranu spoločnú s pôvodným n-uholníkom? Z opačného uhla pohľadu sa dá tá istá veta povedať tak, že trojuholník má najviac dve hrany spoločné s inými trojuholníkmi. Teda každý trojuholník susedí najviac s dvoma ďalšími. S trochou počítania si vieme dokonca zrátať, že (okrem prípadu, keď $n = 3$ a celý n-uholník je jeden nerozdelený trojuholník) máme práve $n-4$ trojuholníkov s dvoma susedmi a dva trojuholníky majú jedného suseda.

Toto pozorovanie nám stačí na to, aby sme si všimli, že trojuholníky tvoria akoby dlhého trojuholníkového „hada“. Na jednom konci tvorí trojuholník s jedným susedom hlavu, potom nasleduje telo hada, tvorené trojuholníkmi s dvoma susedmi a chvost hada je zasa trojuholník s jedným susedom. Lepšie je to asi vidieť na obrázku nižšie, kde je ukážka nejakého rozdelenia 8-uholníka, kde hlava je označená písmenom H, a chvost Ch.



Podobného hada nájdete, keď si skúsite v ľubovoľnom správne rozdelenom n-uholníku pospájať susedné trojuholníky. (Hadovi sa odborne hovorí duálny graf. Pre ľubovoľnú trianguláciu je jej duálnym grafom nejaký strom. Vrcholy, kde sa strom vetví, zodpovedajú práve trojuholníkom, ktoré nemajú ani jednu stranu na obvode n-uholníka.)

Pri pohľade na obrázok by nám mohol skrsnúť nápad, že samotné trojuholníky nie sú vôbec dôležité a dôležitý je len ich počet, resp. dĺžka hada. Aby sme si túto hypotézu overili, skúsme sa pozrieť, čo sa stane, keď spravíme nejaký ťah, teda ofarbíme nejakú hranu.

Na začiatku hry máme jeden veľký neofarbený n-uholník zložený z $n-2$ neobsadených trojuholníkov. Očíslujme si ich od 1 (hlava) po $n-2$ (chvost). V prvom ťahu môže Usamec ofarbiť buď nejakú stranu n-uholníka, alebo nejakú uhlopriečku.

Ofarbením strany n-uholníka sme obsadili niektorý jeden trojuholník, nech je i-ty v poradí. Keď teraz odstránime obsadené hrany (zvyšné dve strany i-teho trojuholníka), dostaneme vo všeobecnosti dve nezávislé časti hracieho plánu – ofarbovanie hrán v jednej z nich nijako neovplyňuje situáciu v druhej.

Navyše tie nové časti sú skoro mnohouholníky. Jediný rozdiel oproti pôvodnému n-uholníku je v tom, že z hlavového a tiež z chvostového trojuholníka môže chýbať jedna strana na obvode. To nám ale vôbec nevadí, lebo tieto trojuholníky majú na obvode 2 strany, z ktorých sa aj tak môže ofarbiť len jedna a je jedno, ktorá to bude.

Tak či tak, ofarbením nejakej strany n-uholníka, ktorá je zároveň stranou i-teho trojuholníka, sa nám pôvodný had dĺžky $n-2$ rozpadne na dva hady dĺžok $i-1$ a $n-2-i$.

Pokiaľ by sme neofarbili stranu n-uholníka ale nejakú jeho uhlopriečku, obsadili by sme tak dva susedné trojuholníky. Ak je to i-ty a $i+1$-vý, rozpadne sa pôvodný had na dvoh hadov s dĺžkami $i-1$ a $n-3-i$.

Všimnime si, že v každom momente vieme ľubovoľného hada rozseknúť na ľubovoľnom mieste. Čiže priebeh hry vôbec nezávisí od toho, ako Maru na začiatku nakreslila uhlopriečky, len od čísla n. Zrazu sa nám hra veľmi zjednodušila.

Stratégia:

Zopakujme si teda ako vyzerá hra. Na začiatku máme jedného hada dĺžky $n-2$. Usamec a Maru sa striedajú v ťahoch (Usamec začína). Ťah spočíva v tom, že si vyberieme nejakého hada, ktorý má nenulovú dĺžku k a rozsekneme ho na dve časti. Tieto časti môžu mať ľubovolné nezáporné dĺžky, ale súčet ich dĺžok musí byť $k-1$ alebo $k-2$. (V n-uholníkovej hre môžu obaja hráči samozrejme robiť aj iné ťahy, ale na ľubovoľný ťah, ktorý nepredstavujú rozseknutie hada, protihráč odpovie dokončením červeného trojuholníka.)

Takáto hra sa dá riešiť nejakým všeobecným veľkým kladivom, napríklad Grundyho číslami, no my si vôbec nepotrebujeme komplikovať život. Stačí si všimnúť jednu veľmi jednoduchú vyhrávajúcu stratégiu pre Usamca.

Usamec v prvom ťahu rozsekne hada na dve rovnako dlhé časti. (Teda ak mal pôvodný had nepárnu dĺžku, Usamec obsadí stredný trojuholník, inak obsadí stredné dva.) Následne symetricky opakuje Maruine ťahy: ak Maru spraví nejaký ťah v jednej časti, Usamec spraví analogický ťah v druhej.

Takto je zaručené, že po každom Maruinom ťahu bude vedieť Usamec tiež spraviť ťah. (Ak Maru rozsekla nejakého hada, tak to znamená, že v symetrickej časti existuje rovnaký had, ktorého môže Usamec rozseknúť rovnakým spôsobom.) Preto postupom času (každým ťahom sa zmenší celková dĺžka hadov, takže hra môže trvať najviac n ťahov) Maru nebude môcť spraviť žiaden ťah – všetky hady budú mať nulovú dĺžku. Vtedy bude Maru musieť ofarbiť nejakú hranu už obsadeného trojuholníka, no a v nasledujúcom ťahu Usamec vyhrá.

Implementácia:

Celkom príjemne sa to celé programuje, ak si nejakým spôsobom zabezpečíme prevod medzi pôvodnou hrou a sekaním hada. Hodí sa nám napríklad funkcia, ktorej keď zadáme, stranu n-uholníka alebo uhlopriečku, vráti číslo trojuholníka(ov), do ktorého patrí. Tiež sa zíde opačná funkcia, ktorá pre daný trojuholník povie nejakú jeho neobsadenú stranu.

Časová zložitosť samotného algoritmu bude $O(n)$ na celú hru. (Predpočítanie a prvý ťah zaberú čas $O(n)$ a každý ďalší ťah zaberie čas $O(1)$.) Pamäťová zložitosť bude tiež $O(n)$, keďže si potrebujeme pamätať hrací plán a aktuálny stav hry.

Naša implementácia uvedená nižšie je trochu „ťažkotonážna“. V časovej zložitosti má (kvôli použitiu usporiadaných množín a máp) navyše logaritmický faktor. Namiesto čísel pracujeme všade, kde sa dá, priamo so samotnými objektami – hranami a trojuholníkmi. Aj takto sa to dá robiť :-)

Listing programu (C++)

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

/*
 * mame pas trojuholnikov
 * ten si mozeme predstavit nasledovne:
 *
 * |_|_|_|_|_|_|
 *
 * kde | su v principe uhlopriecky a _ su strany,
 * kazde |_| predstavuje jeden trojuholnik
 * v tomto poradi si hrany ocislujeme:
 *
 * |_|_|_|_|_|_|
 * 0123456789abc
 *
 * teraz vieme lahko tahat symetricky
 * + ku kazdemu trojuholniku si pamatat kolko uz ma cervenych hran
 */

struct hrana {
    int x,y;
    hrana(int a=-1, int b=-1) : x( min(a,b) ), y( max(a,b) ) {}
};
bool operator< (const hrana &A, const hrana &B)
    { if (A.x != B.x) return A.x < B.x; return A.y < B.y; }
bool operator== (const hrana &A, const hrana &B)
    { return (A.x == B.x) && (A.y == B.y); }
bool operator!= (const hrana &A, const hrana &B)
    { return !(A==B); }

struct trojuholnik {
    int x,y,z;
    trojuholnik(int a=-1, int b=-1, int c=-1) {
        x=min(a,min(b,c)); z=max(a,max(b,c)); y=a+b+c-x-z;
    }
};
bool operator< (const trojuholnik &A, const trojuholnik &B) {
    if (A.x != B.x) return A.x < B.x; else
    if (A.y != B.y) return A.y < B.y; else return A.z < B.z;
}
bool operator== (const trojuholnik &A, const trojuholnik &B)
    { return (A.x == B.x) && (A.y == B.y) && (A.z == B.z); }
bool operator!= (const trojuholnik &A, const trojuholnik &B)
    { return !(A==B); }

// pocet vrcholov mnohouholnika
static int N;
// hrana_na_id[h] je poradie hrany h vo vyssie popisanom ocislovani
static map<hrana,int> hrana_na_id;
// toto je to spominane poradie hran
static vector<hrana> poradie;
// a toto je jemu zodpovedajuce poradie trojuholnikov
static vector<trojuholnik> poradie_trojuholnikov;
// navyse si ku kazdemu z nich pamatame, kolko uz ma cervenych hran
static vector<int> cervenych_hran_v_trojuholniku;

// mnozina cervenych hran; pouzijeme ju len pri uplne poslednom tahu
```

```
static set<hrana> cervene_hrany;
static hrana posledny_tah_maru;
// ak uz existuje trojuholnik, ktory vieme dokoncit, toto je on
static trojuholnik vyherny_trojuholnik;

inline int prevv(int x) { return ((x+N-1)%N); }
inline int nextv(int x) { return ((x+1)%N); }

void hraci_plan(int cN, int U[][2]) {
    N = cN;
    // spravime si mnozinu uhlopriecok a mnozinu vsetkych hran
    set<hrana> uhlopriecky;
    for (int n=0; n<N-3; ++n) uhlopriecky.insert( hrana(U[n][0]-1,U[n][1]-1) );
    set<hrana> hrany;
    for (int n=0; n<N-3; ++n) hrany.insert( hrana( U[n][0]-1, U[n][1]-1 ) );
    for (int n=0; n<N; ++n) hrany.insert( hrana( n, nextv(n) ) );

    // pre kazdu uhlopriecku najdeme trojuholniky na jej stranach
    set<trojuholnik> T1;
    for (int n=0; n<N-3; ++n) {
        int u = U[n][0]-1, v = U[n][1]-1;
        if (hrany.count(hrana(prevv(u),v))) T1.insert(trojuholnik(u,v,prevv(u)));
        if (hrany.count(hrana(nextv(u),v))) T1.insert(trojuholnik(u,v,nextv(u)));
        if (hrany.count(hrana(prevv(v),u))) T1.insert(trojuholnik(u,v,prevv(v)));
        if (hrany.count(hrana(nextv(v),u))) T1.insert(trojuholnik(u,v,nextv(v)));
    }

    // pre kazdy trojuholnik najdeme uhlopriecky ktore obsahuje
    // zaroven pre kazdu uhlopriecku najdeme trojuholniky ktore ju obsahuju
    map<trojuholnik,vector<hrana> > uhlopriecky_trojuholnika;
    map<hrana, vector<trojuholnik> > trojuholniky_uhlopriecky;
    for (auto t : T1) {
        if (uhlopriecky.count(hrana(t.x,t.y))) {
            uhlopriecky_trojuholnika[t].push_back(hrana(t.x,t.y));
            trojuholniky_uhlopriecky[ hrana(t.x,t.y) ].push_back(t);
        }
        if (uhlopriecky.count(hrana(t.x,t.z))) {
            uhlopriecky_trojuholnika[t].push_back(hrana(t.x,t.z));
            trojuholniky_uhlopriecky[ hrana(t.x,t.z) ].push_back(t);
        }
        if (uhlopriecky.count(hrana(t.y,t.z))) {
            uhlopriecky_trojuholnika[t].push_back(hrana(t.y,t.z));
            trojuholniky_uhlopriecky[ hrana(t.y,t.z) ].push_back(t);
        }
    }

    // najdeme trojuholnik ktory ma len jedneho suseda
    trojuholnik curt(-1,-1,-1);
    for (auto t : T1) if (uhlopriecky_trojuholnika[t].size()==1u) {curt=t;break;}
    if (N==3) curt = trojuholnik(0,1,2);

    // od tohto trojuholnika postupne prejdeme vsetky, vyplnime hrany do poradia
    hrana posledna_uhlopriecka(-1,-1);
    while (true) {
        {hrana h(curt.x,curt.y); if (!uhlopriecky.count(h)) poradie.push_back(h);}
        {hrana h(curt.x,curt.z); if (!uhlopriecky.count(h)) poradie.push_back(h);}
        {hrana h(curt.y,curt.z); if (!uhlopriecky.count(h)) poradie.push_back(h);}
        poradie_trojuholnikov.push_back(curt);

        hrana dalsia_uhlopriecka(-1,-1);
        {hrana h(curt.x,curt.y);
         if (uhlopriecky.count(h) && h!=posledna_uhlopriecka) dalsia_uhlopriecka=h;}
        {hrana h(curt.x,curt.z);
         if (uhlopriecky.count(h) && h!=posledna_uhlopriecka) dalsia_uhlopriecka=h;}
```

```
        {hrana h(curt.y,curt.z);
         if (uhlopriecky.count(h) && h!=posledna_uhlopriecka) dalsia_uhlopriecka=h;}

        if (dalsia_uhlopriecka == hrana(-1,-1)) {
            // sme na konci
            break;
        } else {
            // ideme na dalsi trojuholnik
            poradie.push_back(dalsia_uhlopriecka);
            posledna_uhlopriecka = dalsia_uhlopriecka;
            for (auto t : trojuholniky_uhlopriecky[dalsia_uhlopriecka])
                if (t != curt) { curt=t; break; }
        }
    }

    for (int n=0; n<2*N-3; ++n) hrana_na_id[poradie[n]]=n;
    posledny_tah_maru = hrana(-1,-1);
    vyherny_trojuholnik = trojuholnik(-1,-1,-1);
    cervenych_hran_v_trojuholniku.resize(N-2,0);
}

void zapln_trojuholnik(int x) {
    ++cervenych_hran_v_trojuholniku[x];
    if (cervenych_hran_v_trojuholniku[x] == 2)
        vyherny_trojuholnik = poradie_trojuholnikov[x];
}

void zacerven_hranu(const hrana &h) {
    cervene_hrany.insert(h);
    int id = hrana_na_id[h];
    if (id % 2) zapln_trojuholnik(id/2); else {
        if (id/2>0) zapln_trojuholnik(id/2 - 1);
        if (id/2<N-2) zapln_trojuholnik(id/2);
    }
}

void tah_usamca (int hr[2]) {
    if (posledny_tah_maru == hrana(-1,-1)) {
        // prvy tah hry ide do stredu
        zacerven_hranu( poradie[N-2] );
        hr[0] = poradie[N-2].x+1;
        hr[1] = poradie[N-2].y+1;
        return;
    }
    if (vyherny_trojuholnik != trojuholnik(-1,-1,-1)) {
        // uz vieme vyhrat
        trojuholnik t = vyherny_trojuholnik;
        {hrana h(t.x,t.y);
         if (!cervene_hrany.count(h)) { hr[0]=h.x+1; hr[1]=h.y+1; return; }}
        {hrana h(t.x,t.z);
         if (!cervene_hrany.count(h)) { hr[0]=h.x+1; hr[1]=h.y+1; return; }}
        {hrana h(t.y,t.z);
         if (!cervene_hrany.count(h)) { hr[0]=h.x+1; hr[1]=h.y+1; return; }}
    }
    // tahame symetricky s poslednym tahom Maru
    hrana h = poradie[ 2*N - 4 - hrana_na_id[ posledny_tah_maru ] ];
    zacerven_hranu(h);
    hr[0]=h.x+1; hr[1]=h.y+1;
}

void tah_maru (int hr[2]) {
    posledny_tah_maru = hrana( hr[0]-1, hr[1]-1 );
    zacerven_hranu( posledny_tah_maru );
}
```

Výsledky krajských kôl kategórie B

V kategórii B súťaž končí krajským kolom. Na nasledujúcich stranách uvádzame výsledky tohto kola v jednotlivých krajoch. Výsledky neboli koordinované, je teda možné, že v rôznych krajoch boli použité mierne odlišné bodovacie škály. Úspešnými riešiteľmi tohto krajského kola sú tí riešitelia, ktorí získali aspoň 10 bodov.

Banskobystrický kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Dávid Barbora	2 Gym. Františka Švantnera Nová Baňa	3		6	10	19

Bratislavský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Marián Longa	2 Škola pre mim. nadané deti BA	6		3	9	18
2. Ondrej Bohdal	2 Gym. Jura Hronca BA	2	4	4	5	15

Košický kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Peter Kőváry	2 Gym. Poštová Košice	10	9	8	10	37
2. Róbert Schönfeld	2 Gym. Poštová Košice	6	2	4	10	22
3. Matúš Maďar	1 Gym. P. Horova Michalovce	1	0	1	8	10
4. Marek Ceľuch	-1 Gym. Sečovce	2	0	0	7	9
5. Július Čižmár	1 Gym. Sečovce	2	2	1	2	7
6. Martin Vaško	2 Gym. P. Horova Michalovce	0	0	0	3	3

Nitriansky kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Michal Bali	2 Gym. Párovská Nitra	10	5	1	10	26
2. Michal Porubský	1 SKŠ Nitra, Gym. sv. Cyrila a Metoda	8	7	3	2	20
3. Máté Nagy	1 SPŠ Komárno	3	7		3	13
4. Boris Toman	2 Gym. Golianova Nitra	3			8	11
Peter Páleník	2 Gym. Piaristická Nitra	3		3	5	11
6. Marián Pochyba	2 Gym. Golianova Nitra	2			6	8
7. Adam Trtík	2 Gym. Golianova Nitra	2			4	6
8. Tomáš Novák	2 Gym. Golianova Nitra	0	0	1	0	1

Prešovský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Peter Gábor	2 Gym. Konštantínova Prešov	8	2	5	10	25
2. Jaroslav Kravec	2 SPŠE Prešov	3	4	4	4	15
3. Patrik Benyak	1 Gym. Kežmarok	1	2	4	4	11
Slavomír Hanzely	1 Gym. Sabinov	2	2	3	4	11

Trenčiansky kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Adam Mečiar	2 Gym. Nedožerského Prievidza	5	8	9	6	28
2. Adam Hlaváč	1 Gym. Nedožerského Prievidza	2	4	2	6	14
3. Filip Ayazi	2 Gym. Ľudovíta Štúra Trenčín	3		4	6	13
4. Michal Staník	-2 ZŠ Kubranská Trenčín	3		0	8	11
5. Jakub Arbet	-2 ZŠ Kubranská Trenčín	2		3	2	7
6. Roman Dobiáš	2 Gym. Ľudovíta Štúra Trenčín	3	0	3	0	6

V Trnavskom ani Žilinskom kraji sa krajské kolo kategórie B neuskutočnilo, keďže doň z domáceho kola nik nepostúpil.

Výsledky celoštátneho kola kategórie A

Celoštátne kolo kategórie A sa v 28. ročníku Olympiády v informatike uskutočnilo v dňoch 20. až 23. marca v Košiciach, na pôde Prírodovedeckej fakulty Univerzity Pavla Jozefa Šafárika. Najlepších pätnásť riešiteľov bolo vyhlásených za úspešných a z nich najlepších sedem za víťazov celoštátneho kola.

Meno	Ročník, škola	1.	2.	3.	4.	5.	Σ
1. Eduard Batmendijn	2. Cirk. gym. Stará Ľubovňa	8	9	9	15	15	56
2. Jakub Šafin	4. Gym. P. Horova Michalovce	10	10	10	15	1	46
3. Jozef Marko	4. Gym. Lettricha Martin	6	7	10	14	1	38
4. Jaroslav Petrucha	4. Gym. Metodova BA	6	5	10	15	0	36
5. Jerguš Greššák	4. Škola pre mim. nad. deti BA	1	1	8	5	15	30
6. Michal Bui Truc Lam	2. Gym. Grösslingová BA	6	4	10	9	0	29
Mário Lipovský	3. Gym. Jura Hronca BA	6	4	10	6	3	29
8. Askar Gafurov	4. Gym. Grösslingová BA	6	8	4	5	4	27
9. Lukáš Ivan	3. Gym. Jura Hronca BA	2	5	10	4	4	25
Pavel Madaj	2. Gym. Nedožerského Prievidza	2	8	10	5	0	25
11. Vladimír Macko	4. Gym. Ľ. Štúra Zvolen	2	7	8	7	0	24
12. Barbora Kováčová	3. Škola pre mim. nad. deti BA	2	8	6	6	0	22
13. Michal Korbela	3. Gym. Bánovce nad Bebravou	0	8	9	4	0	21
14. Richard Molnár	4. Ev. gym. Lipt. Mikuláš	2	4	8	5	1	20
Viktória Vozárová	3. Gym. Jura Hronca BA	4	7	4	4	1	20
16. Norbert Slivka	3. Gym. Tajovského B. Bystrica	1	5	10	3	0	19
17. Miroslav Kravec	4. SPŠE Prešov	4	4	4	6	0	18
Kamila Součková	4. Škola pre mim. nad. deti BA	1	8	5	4	0	18
19. Róbert Eckhaus	3. Gym. Konštantínova Prešov	0	5	5	6	0	16
Michal Smolík	4. Gym. Grösslingová BA	2	2	8	4	0	16
21. Matej Badin	3. Gym. Jura Hronca BA	1	5	5	2	2	15
22. Martin Kalužník	4. GLN Tomášikova BA	2	2	4	5	1	14
Rastislav Rabatin	4. Gym. Jura Hronca BA	1	5	4	4	0	14
24. Filip Pokrývka	3. Gym. Bánovce nad Bebravou	2	2	4	5	0	13
25. Filip Matušák	3. Gym. Námestovo	0	2	5	5	0	12
26. Michal Bock	4. Gym. Grösslingová BA	2	1	4	4	0	11
27. Viktor Jančík	3. Gym. Kukučínova Poprad	0	1	3	5	0	9
28. Emanuel Tesař	1. Gym. B. S. Timravy Lučenec	0	1	1	5	0	7
29. Miroslav Stankovič	3. Gym. Poštová Košice	0	1	4	0	0	5

Výsledky výberového sústredenia

V dňoch 28. apríla až 4. mája 2013 sa v Bratislave konalo výberové sústredenie. O účasť na medzinárodných akciách na ňom bojovalo najlepších 13 riešiteľov a riešiteliek celoštátneho kola OI-A. Ako každý rok, najlepší štyria slovenskí riešitelia sa kvalifikovali na Medzinárodnú olympiádu v informatike. Na základe výberového sústredenia taktiež SK OI vybrala reprezentačný tím pre Stredoeurópsku olympiádu v informatike a pre prípravné Česko-poľsko-slovenské stretnutie.

V nasledujúcej tabuľke sú uvedené výsledky výberového sústredenia. Stĺpec „štart“ obsahuje body, s ktorými súťažiaci začínali (t.j. súčet bodov za celoštátne kolo OI a domáce úlohy).

	Meno	$\sum$	štart	ne	po	ut	st	št	pi	so
1.	Eduard Batmendijn	784.86	90.00	60.0	136.0	111.50	137.36	61.0	135.0	54.0
2.	Jakub Šafin	725.88	103.38	22.5	105.0	125.50	150.00	63.5	106.0	50.0
3.	Jozef Marko	567.08	85.75	44.0	51.0	96.00	114.33	31.0	103.0	42.0
4.	Jaroslav Petrucha	505.89	76.50	30.0	85.5	66.50	100.89	14.0	104.0	28.5
5.	Mário Lipovský	446.86	63.25	21.5	85.0	56.00	80.11	12.0	89.0	40.0
6.	Vladimír Macko	441.76	60.69	30.0	81.5	83.50	59.57	6.0	89.0	31.5
7.	Bui Truc Lam	363.53	59.12	6.5	60.5	36.75	85.16	29.0	69.5	17.0
8.	Jerguš Greššák	340.00	44.00	44.0	52.0	51.00	51.50	8.0	66.0	23.5
9.	Barbora Kováčová	331.59	45.34	0.0	57.5	46.00	56.25	15.5	84.5	26.5
10.	Askar Gafurov	330.09	73.28	36.5	53.5	58.00	77.31	13.5	18.0	—
11.	Lukáš Ivan	314.00	33.00	15.5	33.5	41.00	95.00	13.0	80.0	3.0
12.	Michal Korbela	261.94	52.44	15.5	34.5	44.00	8.50	15.5	57.0	34.0
13.	Pavel Madaj	255.26	26.88	8.0	32.5	46.50	40.39	14.5	55.0	31.5

Do riešenia súťažných úloh sa (aspoň v niektoré dni) zapojili aj ďalší riešitelia zo Slovenska, Čiech a Švajčiarska. Ich výsledky uvádzame v samostatnej tabuľke.

Meno	$\sum$	štart	ne	po	ut	st	št	pi	so
Michal Anderle (SVK)	640.49	100.00	60.00	136.00	131.50	134.99	20.00	20.00	38.00
Štepán Šimsa (CZE)	436.16	100.00	—	—	80.50	113.16	31.00	91.00	20.50
Nikola Djokić (SUI)	367.70	100.00	22.00	—	—	132.70	60.00	53.00	—
Filip Hlásek (CZE)	346.50	100.00	—	—	—	110.00	56.50	80.00	—
Fabian Lyck (SUI)	322.17	100.00	6.50	60.00	29.00	62.17	21.00	21.00	22.50
Timon Stampfli (SUI)	206.23	100.00	0.00	9.00	7.00	42.73	7.00	21.00	19.50

Medzinárodné prípravné sústredenie v Davose

Vďaka blízkej spolupráci medzi národnými olympiádami v informatike na Slovensku a vo Švajčiarsku mali aj tento rok vybraní riešitelia KSP a OI možnosť zúčastniť sa prípravného sústredení vo švajčiarskom Davose, a to v dňoch 11. až 16. februára 2013. Sústredenía sa tiež zúčastnili delegácie z Rumunska a Ruska. V prvej tabuľke uvádzame výsledky dlhodobej súťaže, ktorá prebiehala počas celého sústredenia.

Meno	kraj.	day1	day2	day3	day4	$\sum$
1. Radu Stefan Voroneanu	ROM	320	400	394	302	1416
2. Vlad Alexandru Gavrila	ROM	320	400	306	332	1358
3. Eduard Batmendijn	SVK	320	400	206	300	1226
4. Jakub Šafin	SVK	320	400	152	270	1142
5. Darius Rares Buhai	ROM	320	400	164	258	1142
6. Johannes Kapfhammer	SUI	200	300	114	250	864
7. Vsevolod Stepanov	RUS	200	225	130	250	805
8. Jozef Marko	SVK	200	186	90	246	722
9. Igor Labutin	RUS	300	120	58	212	690
10. Evgeniy Zamyatin	RUS	240	100	144	204	688
11. Lukas Roth	SUI	120	70	66	210	466
12. Cedric Neukom	SUI	154	66	12	200	432
13. Benjamin Schmid	SUI	180	62	16	120	378
14. Fabian Lyck	SUI	154	78	52	88	372
15. Cedric Münger	SUI	110	50	16	190	366
16. Pascal Sommer	SUI	100	50	14	190	354
17. Lorenz Brun	SUI	70	84	34	125	313
18. Stephan Leuch	SUI	100	75	16	106	297
19. Timon Stampfli	SUI	80	50	14	150	294
20. Timo Bräm	SUI	100	87	14	82	283
21. Florian Schroeder	SUI	22	50	18	115	205

V druhej tabuľke uvádzame výsledky jednokolovej súťaže „Davos Cup“. Rovnaké úlohy riešilo aj 105 stredoškolákov v Rusku, poradie v zátvorke je poradím v spoločnej výsledkovej listine.

		Meno	kraj.	úl.1	úl.2	úl.3	úl.4	$\sum$
1.	(2.)	Vlad Alexandru Gavrila	ROM	100	75	100	100	375
2.	(7.)	Jakub Šafin	SVK	100	55	10	100	265
3.	(14.)	Radu Stefan Voroneanu	ROM	100	50	35	52	237
4.	(17.)	Igor Labutin	RUS	100	30	100	—	230
5.	(19.)	Johannes Kapfhammer	SUI	100	0	100	26	226
6.	(20.)	Darius Rares Buhai	ROM	100	0	100	20	220
7.	(22.)	Evgeniy Zamyatin	RUS	100	30	55	26	211
8.	(24.)	Vsevolod Stepanov	RUS	100	0	85	24	209
9.	(31.)	Fabian Lyck	SUI	100	5	55	24	184
10.	(33.)	Eduard Batmendijn	SVK	—	65	100	8	173
11.	(43.)	Benjamin Schmid	SUI	70	45	—	24	139
12.	(44.)	Timon Stampfli	SUI	80	0	30	24	134
13.	(77.)	Lukas Roth	SUI	60	5	—	22	87
14.	(85.)	Cedric Münger	SUI	60	0	—	2	62
15.	(96.)	Timo Bräm	SUI	40	—	—	—	40
16.	(99.)	Florian Schroeder	SUI	—	15	—	24	39
17.	(104.)	Jozef Marko	SVK	—	—	—	24	24
		Pascal Sommer	SUI	—	0	—	24	24
19.	(115.)	Lorenz Brun	SUI	10	—	—	—	10
20.	(119.)	Stephan Leuch	SUI	—	—	—	6	6
21.	(120.)	Cedric Neukom	SUI	—	0	—	—	0

Česko-poľsko-slovenské prípravné sústredenie

Pätnáste súťažné stretnutie najlepších stredoškolákov z Čiech, Poľska a Slovenska sa uskutočnilo v poľskom hlavnom meste Varšave v dňoch 16. až 22. júna 2013. Slovenskú výpravu sprevádzali Michal Anderle a Marián Horňák, obaja študenti FMFI UK. Víťazom súťaže sa síce opäť stal súťažiaci z Poľska, avšak na rozdiel od niekoľkých predchádzajúcich rokov sa tentokrát na popredných pozíciách umiestnili aj viacerí súťažiaci z Čiech a Slovenska. Podrobné výsledky uvádzame v priloženej tabuľke.

meno	kraj.	day1	day2	day3	day4	$\sum$
1. Karol Farbiś	POL	206	180	242	200	828
2. Krzysztof Pszeniczny	POL	148	200	270	200	818
3. Eduard Batmendijn	SVK	122	112	206	235	675
4. Štěpán Šimsa	CZE	148	70	210	215	643
5. Błażej Magnowski	POL	96	40	240	200	576
6. Martin Raszyk	CZE	48	130	154	200	532
7. Marek Sommer	POL	108	100	190	130	528
8. Jakub Šafin	SVK	160	100	128	100	488
9. Mário Lipovský	SVK	96	0	182	200	478
10. Jaroslav Petrucha	SVK	48	100	128	200	476
11. Jan-Sebastian Fabík	CZE	72	0	182	170	424
12. Stanisław Dobrowolski	POL	148	10	100	155	413
13. Bui Truc Lam	SVK	48	0	128	205	381
14. Jan Ludziejewski	POL	48	80	36	160	324
15. Ondřej Hlavatý	CZE	40	0	72	120	232
16. Michal Punčochář	CZE	6	0	44	40	90
17. Martin Hora	CZE	6	0	24	0	30

Stredoeurópska olympiáda v informatike

V roku 2013 sa Stredoeurópska olympiáda v informatike (CEOI) konala v Chorvátsku v meste Primošten. Z organizačných dôvodov bol jej termín konania netradične neskorý – až od 13. do 19. októbra 2013. Súťaže sa zúčastnili tímy členských krajín CEOI: Česka, Slovenska, Poľska, Maďarska, Nemecka, Rumunska a dva tímy z domáceho Chorvátska. Okrem nich ešte súťažili aj pozvané tímy zo Slovinska a Švajčiarska.

Slovensko na CEOI 2013 reprezentovali štyria stredoškoláci: Eduard Batmendijn (Cirk. gym. Stará Ľubovňa), Michal Bui Truc Lam (Gym. Grösslingová Bratislava), Barbora Kováčová (ŠPMNDaG Bratislava) a Mário Lipovský (Gym. Jura Hronca Bratislava). Barbora sa o svojej účasti na súťaži dozvedela až 24 hodín pred odchodom, keďže na CEOI postúpila ako náhradník za Vladimíra Macka, ktorý sa súťaže nemohol zúčastniť zo zdravotných dôvodov.

Našu delegáciu na tejto súťaži viedli doc. RNDr. Gabriela Andrejková, CSc. (PF UPJŠ v Košiciach) a prof. Ing. Veronika Stoffová, CSc. (KI PF UJS v Komárne). Ako pozorovateľ sa CEOI 2013 navyše zúčastnil Michal Anderle (študent FMFI UK v Bratislave).

Len po druhý raz v celej histórii CEOI (od roku 1994) sa podarilo, že celkovým víťazom sa stal súťažiaci zo Slovenska. K doteraz jedinému Petrovi Bellovi, ktorému sa to podarilo v roku 2002, sa teraz pridal Eduard Batmendijn. Okrem neho nás potešila aj Barbora Kováčová ziskom bronzovej medaile.

Podrobné výsledky našich súťažiacich uvádzame v tabuľke.

poradie a meno	1. deň			2. deň			$\sum$	medaila
1. Eduard Batmendijn	100	25	30	100	0	100	355	zlato
14. Barbora Kováčova	48	45	0	30	25	40	188	bronz
26. Mário Lipovský	44	45	0	10	0	40	139	–
37. Michal Bui Truc Lam	10	45	–	10	25	–	90	–

Medzinárodná olympiáda v informatike

Slovenská republika získala na 25. Medzinárodnej olympiáde v informatike v austrálskom Brisbane dve zlaté a po jednej striebornej a bronzovej medaile.

Slovensku sa tak po dlhšom čase (dve zlaté sme mali naposledy v roku 2005) podarilo výraznejšie presadiť v medzinárodnej konkurencii. Najúspešnejšou krajinou sa stala Čína so štyrmi zlatými medailami. Ďalšie tri zlaté medaily si odniesli Rusi, po dve potom okrem Slovenska ešte USA, Kórea a Rumunsko. Ďalších 10 krajín získalo po jednej zlatej medaile. Podarilo sa nám zdolať najväčších konkurentov z nášho regiónu – Poliakov aj Čechov, ktorí tento rok odchádzajú bez zlatej medaily.

Zlatú medailu z minulého roku obhájil Eduard Batmendijn z Cirkevného gymnázia sv. Mikuláša v Starej Ľubovni, ktorý skončil so 485 bodmi (zo 600 možných) na 22. mieste ako šiesty európan v poradí. Druhé zlato patrí Jakubovi Šafinovi z Gym. P.Horova v Michalovciach za 24. miesto. Ďalšími našimi medailistami sú Jozef Marko z Gym. Lettricha v Martine (striebro, 62. miesto) a Jaroslav Petrucha z Gym. Metodova v Bratislave (bronz, 109. miesto).

Zvyšok slovenskej výpravy tvorili člen medzinárodného odborného výboru súťaže RNDr. Michal Forišek, PhD. (FMFI UK), vedúci delegácie RNDr. Rastislav Krivoš-Belluš, PhD. (PF UPJŠ), pedagogický vedúci Mgr. Vladimír Boža (FMFI UK) a ako autorka jednej zo súťažných úloh Mgr. Monika Steinová, PhD. (Zürich, Švajčiarsko)

Tohto roku sa Medzinárodnej olympiády v informatike zúčastnilo 299 oficiálnych súťažiacich zo 77 krajín celého sveta, ktorí prešli sitom národných súťaží. Absolútnym víťazom sa stal Číňan Lijie Chen s počtom bodov 569, pričom v piatich úlohách zo šietich získal maximum, teda po 100 bodov.

Súčasťou programu mimo súťažných dní bola aj prehliadka austrálskej zoo s voľne poskakujúcimi kengurami, ako aj pláže Golden Coast, kde je aj teraz v ich zime možné okúpať sa či zasurfovať si.

Kompletné výsledky našich súťažiacich uvádzame v nasledujúcej tabuľke.

Meno	1. deň			2. deň			$\sum$	medaila
Eduard Batmendijn	100	67	55	100	100	63	485	22. miesto, zlato
Jakub Šafin	100	91	55	100	100	37	483	24. miesto, zlato
Jozef Marko	59	80	55	100	90	10	394	62. miesto, striebro
Jaroslav Petrucha	47	49	9	76	76	37	294	109. miesto, bronz

IUVENTA – Slovenský inštitút mládeže je príspevková organizácia priamo riadená Ministerstvom školstva, vedy, výskumu a športu SR. Pripravuje a riadi množstvo zaujímavých programov a projektov pre mladých ľudí, pracovníkov s mládežou a ľudí zodpovedných za mládežnícku politiku. Snaží sa o to, aby mladí ľudia poznali svoje možnosti, boli aktívni a pracovali na sebe tak, aby boli raz úspešní a uplatnili sa na trhu práce. IUVENTA vychováva mládež k ľudským právam, podporuje rozvoj dobrovoľníctva, vzdelávacie programy a tiež mladé talenty.

IUVENTA je realizátorom národných projektov v oblasti práce s mládežou KomPrax – Kompetencie pre prax a Praktik – Praktické zručnosti cez neformálne vzdelávanie v práci s mládežou, ktoré sú podporené z Európskeho sociálneho fondu. Zároveň administruje Programy finančnej podpory aktivít detí a mládeže MŠVVaŠ SR ADAM, je národnou agentúrou programu EÚ Mládež v akcii a národným partnerom európskej informačnej siete pre mládež Eurodesk.

Kontakt: IUVENTA – Slovenský inštitút mládeže
Búdková 2
SK-811 04 Bratislava 1
tel.: (421-2) 59 29 61 12
fax: (421-2) 59 29 61 23
e-mail: iuventa@iuventa.sk
web: www.iuventa.sk, www.facebook.com/iuventa

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty ôsmy ročník Olympiády v informatike
Vydavateľ: IUVENTA – Slovenský inštitút mládeže
Rok vydania: 2013
Rozsah: 137 strán
Náklad: 400 výtlačkov
ISBN: 978-80-8072-145-9
Neprešlo jazykovou úpravou