Dvadsiaty štvrtý ročník Olympiády v informatike

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

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty štvrtý ročník Olympiády v informatike
ISBN 978-80-8072-100-8

Obsah

O priebehu 24. ročníka Olympiády v informatike

V školskom roku 2008/09 prebehol na Slovensku už dvadsiaty štvrtý ročník Olympiády v informatike (OI). Podobne ako v minulom ročníku, aj v tomto prebehla súťaž v dvoch kategóriách: A a B.

Do kategórie B, určenej pre prvákov a druhákov klasických štvorročných stredných škôl, sa zapojilo 56 žiakov. Z nich bolo 38 pozvaných na krajské kolá, ktorými súťaž skončila.

Do ťažšej kategórie A, určenej pre všetkých stredoškolákov bez obmedzenia, sa zapojilo 70 žiakov. Najlepších 47 postúpilo do krajského, a z nich najlepších 30 do celoštátneho kola OI. Celoštátne kolo sa v tomto školskom roku konalo v obci Rajecká Lesná v Žilinskom kraji.

Najlepší riešitelia kategórie A boli následne pozvaní na týždňové výberové sústredenie, kde sa určila reprezentácia Slovenska na medzinárodných súťažiach – Stredoeurópskej olympiáde v informatike (CEOI) v Rumunsku a Medzinárodnej olympiáde v informatike (IOI) v Bulharsku.

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, FMFI UK, Bratislava
- Mgr. Juliana Lipková, FMFI UK, Bratislava
- Mgr. Ján Katrenič, PF UPJŠ, Košice
- Mgr. Lukáš Poláček, FMFI UK, Bratislava
- PaedDr. Miloslava Sudolská, PhD.,
  KI FPV UMB, krajská predsedkyňa pre BB
- RNDr. Eva Hanulová, Gym. J. Hronca, krajská predsedkyňa pre BA
- RNDr. Rastislav Krivoš-Belluš,
  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
- Mgr. Blanka Thomková, Gym. Jána Hollého, krajská predsedkyňa pre TT
- RNDr. Peter Varša, PhD., KI FRI ŽU, krajský predseda pre ZA

Zástupcom IUVENTY zabezpečujúcim organizačnú stránku súťaže bol už tradične Ing. Tomáš Lučenič.

Hoci je Olympiáda v informatike už tri roky samostatnou súťažou, naďalej úzko spolupracuje ako s Matematickou olympiádou na Slovensku, tak aj s Matematickou olympiádou, kategorie P (programování) v susednej Českej republike. Úlohy našej kategórie A sú spoločné so súťažou v ČR a pripravujú ich striedavo organizátori z oboch krajín.

Viacerí bývalí organizátori slovenskej Olympiády v informatike (spomeňme napr. RNDr. Richarda Kráľoviča, PhD. a Mgr. Moniku Steinovú) v posledných rokoch taktiež pomáhajú rozbehnúť organizáciu informatickej olympiády vo Švajčiarsku. S organizátormi Poľskej olympiády v informatike spolupracujeme každoročne pri príprave česko-poľsko-slovenského súťažného stretnutia. Veríme, že táto medzinárodná spolupráca nám pomáha robiť našu súťaž čoraz kvalitnejšou.

V nasledujúcom školskom roku nás okrem bežných „povinností“ čaká aj jedna skúška ohňom – organizácia Stredoeurópskej olympiády v informatike (CEOI 2010). Dúfajme, že všetko pôjde podľa našich predstáv.

Michal Forišek, podpredseda SK OI

Zadania domáceho kola kategórie A

A-I-1 Meníme históriu

Pred veľa, veľa rokmi, keď sa ešte voda sypala a piesok lial, vládol v krajine Siedmeho sveta kráľ Ogrgeľ Veľký. I zaumienil si on, že by bolo treba zmeniť učebnice dejepisu. Tie totiž napísal kráľ Papľuh IV. A samozrejme v nich tvrdí, že on, Papľuh IV., je ten najväčší, najmúdrejší a najurodzenejší. Toto však nie je pravda – Ogrgeľ Veľký je predsa ten, ktorý je najväčší, najmúdrejší a najurodzenejší. Tú veľkosť má dokonca aj v mene! Všetky učebnice by bolo treba zmeniť... Lenže s tým je práca. A keď je on ten najmúdrejší, nebude ju predsa robiť on.

Súťažná úloha:

Kráľ zostavil zoznam dvojíc slov. Každú dvojicu slov tvorí jedno staré nevhodné slovo, a jedno nové vhodné, ktorým treba to staré nahradiť. Napíšte program, ktorý načíta zoznam dvojíc slov a vstupný text, a následne vo vstupnom texte všetky výskyty nevhodných slov nahradí ich novými, vhodnými „ekvivalentmi“.

Formát vstupu:

V prvom riadku súboru sa nachádza prirodzené číslo N ($N < 100\,000$), ktoré udáva počet nevhodných slov. Nasleduje N riadkov, pričom v každom sa nachádzajú vždy dve slova oddelené medzerou. Slová sú tvorené iba z veľkých písmen anglickej abecedy. Prvé slovo v každom z týchto N riadkov je nevhodné, druhé je jeho náhrada. Od riadku $N + 2$ až do konca súboru nasleduje samotný text učebnice. Text je tvorený iba z veľkých písmen anglickej abecedy a medzier (a koncov riadkov), pričom súvislé úseky písmen tvoria jednotlivé slová. Môžete predpokladať, že žiadné slovo nebude dlhšie ako 255 písmen a že žiadne dve nevhodné slová nebudú rovnaké.

Vstupné súbory použité pri testovaní budú mať unixové konce riadkov – každý riadok, vrátane posledného, končí znakom line feed (ASCII hodnota 10).

Formát výstupu:

Výstupom programu je text, v ktorom boli všetky nevhodné slová nahradené ich vhodnejšími ekvivalentmi. Zvyšok súboru musí zostať bez zmeny, vrátane rovnakých medzier a rozdelenia slov do riadkov.

Príklad:

Vstup

```
5
PAPLUH OGRGEL
OGRGEL PAPLUH
DRACICI DRAKOVI
KRK REBRO
ZBYTOCNA DVOJICA
A TAK SA STALO
ZE VELKY PAPLUH
KRKOLOMNE ZLOMIL
DRACICI KRK
 V TEJTO UCEBNICI   BUDE
HANENY OGRGEL

JANOSIKA ZAVESILI
ZA PIATE REBRO
```

Výstup

```
A TAK SA STALO
ZE VELKY OGRGEL
KRKOLOMNE ZLOMIL
DRAKOVI REBRO
 V TEJTO UCEBNICI   BUDE
HANENY PAPLUH

JANOSIKA ZAVESILI
ZA PIATE REBRO
```

Všimnite si, že slová `KRKOLOMNE` *(tretí riadok) ani* `REBRO` *(posledný riadok) sa nezmenili.*

Odovzdávanie riešení:
Toto je praktická úloha. Odovzdávate **len funkčný, odladený program** prostredníctvom webového rozhrania.

A-I-2 Prechádzka po kanáloch

Mnoho rokov prebehlo, odkedy Jean Valjean utekal Parížskymi kanálmi, avšak ani napriek tomu nestratili nič zo svojej majestátnosti a tajomnosti. A dnes sa Charlotte a Jacques rozhodli tieto kanály preskúmať. Charlotte sa ale kanálov bojí, preto ostane sedieť vonku a vysielačkou bude rozprávať s Jacquesom. Ten bude kráčať kanálmi a hľadať artefakty a zaujímavosti (nazvime ich súhrnne *poklad*), ktoré Jean vo svojej mape vyznačil.

Podľa plánov, ktoré našli v tajnom denníku Jeana Valjeana, sú kanály pod Parížom orientované podľa svetových strán a zodpovedajú mape na štvorčekovom papieri. Každý štvorček na mape predstavuje políčko so stranou 1 m. V kanáloch občas bývajú mreže s dvierkami, ktoré sa dajú zamknúť. Každá mreža zaberá jedno políčko. Keď je zamknutá, nedá sa cez zodpovedajúce políčko ísť. Jean našťastie počas svojich potuliek kanálmi našiel a do mapy zakreslil všetky

mreže. A čo je ešte dôležitejšie, získal aj kľúče k nim a poukrýval ich na rôznych miestach v kanáloch. Zámky na mrežiach aj kľúče sú štyroch farieb: červené, modré, zelené a fialové. Keď máme kľúč, vieme ním otvárať zámky jeho farby.

Aby sa Jacques nestratil, chodí krokmi dlhými 1 m a otáča sa len o násobky 90°, t.j. môže sa pohybovať len vodorovne alebo zvisle na mape. Jacques môže na políčko vojsť, ak je voľné, je tam poklad, alebo tam sú mreže, ktoré vie otvoriť. Nemôže vojsť na políčko kde je stena alebo zamknuté mreže, od ktorých nemá kľúč.

Parížska kanalizácia sa nedá opustiť. Je navrhnutá tak, že keď Jacques vyjde z políčka na najľavejšom konci mapy, ocitne sa na políčku v tom istom riadku, ale v najpravejšom stĺpci a opačne. Podobne aj krok z prvého riadku nahor ho premiestni do posledného riadku toho istého stĺpca a naopak. Akonáhle Jacques nájde prvý poklad, tak je Charlottina úloha splnená.

Súťažná úloha:

Napíšte program, ktorý načíta mapu kanalizácie (vrátane Jacquesovej začiatočnej pozície) a nájde najkratšiu cestu vedúcu k niektorému pokladu.

Je možné, že na mape žiaden poklad nie je, prípadne že sa Jacques k žiadnemu nebude vedieť dostať. Taktiež môže byť v kanáloch viac pokladov, viac mreží alebo kľúčov rovnakej farby.

Formát vstupu:

Prvý riadok vstupu obsahuje dve prirodzené čísla R (počet riadkov) a S (počet stĺpcov) oddelené jednou medzerou. Môžete predpokladať, že $1 \leq R \cdot S \leq 1\,000\,000$. Každý z nasledujúcich riadkov obsahuje práve S znakov popisujúcich typy políčok na mape. Existujú tieto možné typy políčok:

`#`	stena
`.`	voľné políčko (chodba, miestnosť)
`*`	Jacquesova začiatočná pozícia – takéto políčko bude práve jedno
`CZMF`	zamknuté červené, zelené, modré alebo fialové mreže
`czmf`	červený, zelený, modrý alebo fialový kľúč
`$`	poklad

Mapa je vo vstupe orientovaná podľa zaužívaných konvencií, t.j. sever je hore.

Formát výstupu:

Výstup má obsahovať jeden riadok a v ňom najkratšiu cestu, ktorou sa Jacques môže dostať k niektorému z pokladov. Ak je rovnako krátkych ciest viac, vypíšte ľubovoľnú z nich. Cestu vypisujte ako postupnosť znakov ‘`S`’, ‘`J`’, ‘`V`’ a ‘`Z`’, ktoré určujú, na ktorú svetovú stranu má Jacques spraviť krok. Ak neexistuje

žiadna postupnosť spĺňajúca zadanie, tak namiesto popisu cesty vypíšte jeden riadok a v ňom jediné slovo '`nemozne`'.

Príklady:

Vstup

```
3 5
#####
#*F$#
#####
```

Výstup

```
nemozne
```

Bez kľúča cez zamknutú fialovú mrežu neprejde.

Vstup

```
3 19
####################
#$F.zMc.*.Cm.Z.ZZf#
####################
```

Výstup

```
ZZVVVVVZZZZZZZVVVVVVV
VVVVVZZZZZZZZZZZZZZZ
```

Jacques postupne vyzdvihne červený, modrý, zelený a fialový kľúč, v tomto poradí.
(Výstup má byť celý v jednom riadku, je zalomený na dva kvôli sadzbe.)

Vstup

```
3 1
*
.
$
```

Výstup

```
S
```

Prejde cez severný okraj.

Odovzdávanie riešení:
Toto je praktická úloha. Odovzdávate **len funkčný, odladený program** prostredníctvom webového rozhrania.

A-I-3 Horská dráha

Na Vyšnej Klondike ešte minulý rok chodil vláčik. Kvôli problémom s rezervačným systémom však skrachoval a ostali len koľajnice.

Tie ležali len tak ladom a hrdzaveli, až kým jedného dňa neprišli opäť raz dvaja podnikaví zlatokopi, Santo a Banto, a nerozhodli sa, že na nich postavia pravú horskú dráhu.

Horská dráha musí začínať aj končiť v niektorej zo staníc železnice. Aby nebola jazda ňou ani príliš nudná, ani príliš nebezpečná, mal by mať jeden jej koniec práve o X metrov vyššiu nadmorskú výšku ako druhý. To sa však samozrejme nemusí dať dosiahnuť.

Súťažná úloha:

Trať sa skladá z $N+1$ staníc a N úsekov medzi nimi. Santo a Banto sa prešli pozdĺž nej a pre každý úsek i zistili hodnotu d_i, o ktorú sa na tomto úseku zmení nadmorská výška. (Ak je d_i kladné, trasa stúpa do kopca, ak je záporné, klesá, a ak je rovné nule, sú začiatok a koniec rovnako vysoko.)

Nájdite dve čísla i a j, pre ktoré je rozdiel nadmorských výšok konca j-teho a začiatku i-teho úseku čo najbližší k číslu X. Inými slovami, v postupnosti $d_1, \ldots, d_N$ nájdite tú súvislú podpostupnosť $d_i, \ldots, d_j$, ktorej súčet je S a hodnota $|X - |S||$ je najmenšia možná. (Stačí nájsť jedno ľubovoľné optimálne riešenie.)

Príklad:

Vstup

```
N=5
X=7
3 5 -2 5 9
```

Výstup

```
2 4
```

Indexom $i = 2$ a $j = 4$ zodpovedá podpostupnosť $5, -2, 5$, ktorej súčet je 8. Žiadna podpostupnosť nemá súčet 7, preto toto je jedna z optimálnych postupností.
Iné správne riešenia sú $i = 1$, $j = 2$ (súčet 8) a $i = 1$, $j = 3$ (súčet 6).

Odovzdávanie riešení:

Toto je teoretická úloha. Riešenie, spĺňajúce požiadavky uvedené v pravidlách, buď pošlite poštou na adresu vašej krajskej komisie, alebo odovzdajte vo formáte PDF prostredníctvom webového rozhrania.

A-I-4 Zásobníkové počítače

V tomto ročníku olympiády sa budeme v teoretickej úlohe stretávať so zásobníkovými počítačmi. V študijnom texte uvedenom za zadaním tejto úlohy sú tieto stroje popísané.

Súťažná úloha:

Reťazec voláme *palindróm*, ak sa číta rovnako odpredu aj odzadu – t. j., jeho prvý znak je rovnaký ako posledný, druhý ako predposledný, atď.

a) (5 bodov)
Napíšte program pre zásobníkový počítač, ktorý o zadanom reťazci rozhodne, či je to palindróm.
Snažte sa dosiahnuť optimálnu časovú zložitosť.

b) (5 bodov)
Napíšte program pre zásobníkový počítač, ktorý o zadanom reťazci rozhodne, či je to palindróm.
Snažte sa použiť minimálny počet zásobníkov.

Odovzdávanie riešení:

Toto je teoretická úloha. Riešenie, spĺňajúce požiadavky uvedené v pravidlách, buď pošlite poštou na adresu vašej krajskej komisie, alebo odovzdajte vo formáte PDF prostredníctvom webového rozhrania.

Študijný text

V tomto ročníku olympiády si predstavíme zásobníkové počítače. Pamäť týchto počítačov je tvorená niekoľkými zásobníkmi. Každý zásobník obsahuje postupnosť hodnôt, s ktorou vie robiť tri operácie: Pridať novú hodnotu na koniec postupnosti (vložiť ju na vrch zásobníka), odstrániť hodnotu z konca postupnosti (vybrať ju zo zásobníka) a zistiť, či je postupnosť v zásobníku prázdna. Okrem zásobníkov vieme mať už len konštantný počet obyčajných premenných. Hodnoty uložené v zásobníkoch aj premenných musia mať vopred určený rozsah, ktorý nezávisí od veľkosti vstupu.

Naše zásobníkové počítače budeme programovať v jazyku Stackal. Ide o odrodu jazyka Pascal, upravenú podľa možností našich počítačov. V nasledujúcich odsekoch popíšeme, v čom sa náš jazyk od klasického Pascalu líši.

Premenné môžu byť týchto typov:

– `boolean` (logická premenná nadobúdajúca hodnoty `true` a `false`),

– `char` (premenná obsahujúca jeden znak),

– `a..b` (celočíselná premenná nadobúdajúca hodnoty od a po b vrátane, pričom $0 \leq a \leq b \leq 100$) a

– `stack of T`, kde `T` je typ iný ako `stack` (zásobník obsahujúci postupnosť hodnôt typu `T`).

Zásobníky sú na začiatku behu programu prázdne, obsah ostatných premenných je nedefinovaný.

Vstupom aj výstupom programu je postupnosť znakov – inými slovami, reťazec. Vstup čítame volaním funkcie `read(c)`, kde `c` je premenná typu `char`. Ak sme ešte vstup nedočítali, táto funkcia uloží do premennej `c` ďalšie písmeno zo vstupu a vráti `true`. V opačnom prípade vráti `false` a hodnotu premennej `c` nezmení. Na výstup zapíšeme znak z premennej `c` volaním procedúry `write(c)`. Na vstupe ani výstupe sa nie je možné vracať ani znaky preskakovať, program musí čítať vstup aj zapisovať výstup zaradom, „zľava doprava“.

Na prácu so zásobníkmi budeme mať špeciálne procedúry a funkcie. Nech `x` je premenná typu `T` a `S` je zásobník obsahujúci hodnoty tohto typu (teda `S` je premenná typu `stack of T`). Volaním procedúry `push(S,x)` vložíme do zásobníka `S` obsah premennej `x`. Funkcia `pop(S)` vyberie hodnotu zo zásobníka a vráti ju ako svoju návratovú hodnotu. Túto funkciu môžeme použiť aj ako procedúru, ak nás vybratá hodnota nezaujíma. Ak je pri volaní funkcie `pop(S)` zásobník `S` prázdny, program skončí s chybou – toto teda robiť nesmieme. Posledná užitočná funkcia je `empty(S)`, ktorá vráti `true` ak je zásobník `S` prázdny a `false` ak nie je. Žiadnym iným spôsobom nevieme s obsahom zásobníkov manipulovať.

K dispozícii máme všetky príkazy klasického Pascalu, môžeme si definovať aj vlastné pomocné procedúry a funkcie, avšak musíme dodržať niekoľko obmedzení. Je zakázané používať rekurziu. Priradzovací príkaz `:=` nesmieme použiť na zásobníky. Zásobník smieme dať ako parameter funkcii len odkazom, teda použitím kľúčového slova `var`.

(Najjednoduchšie riešenie, ako obmedzenia dodržať: Deklarujte všetky použité zásobniky na začiatku programu ako globálne premenné. Potom postupne definujte pomocné funkcie tak, aby každá z nich volala len funkcie definované skôr.)

Časovú a pamäťovú zložitosť definujeme podobne ako u klasických programov. Čas meriame počtom vykonaných príkazov, použitá pamäť je rovná maximálnemu počtu hodnôt, ktoré si program niekedy počas svojho behu pamätal v premenných a zásobníkoch.

Často bude našim cieľom minimalizovať počet použitých zásobníkov, a to aj za cenu horšej časovej zložitosti.

Príklad 1:

Napíšte program, ktorý zistí, či je v zadanom reťazci rovnako veľa písmen `a` a `b`, a podľa toho vypíše na výstup buď znak `1` alebo znak `0`.

Riešenie 1a:

Keďže rozsah celočíselných premenných je obmedzený, nemôžeme si počítať výskyty znakov `a` a `b` v premenných. Môžeme na to však ľahko využiť dva zásobníky. Do jedného si budeme ukladať `a`-čka a do druhého `b`-čka. Po dočítaní vstupu budeme súčasne vyberať znaky z oboch zásobníkov, a zistíme, či sa oba vyprázdnia naraz.

```
var a, b: stack of char;           { dva zasobniky na znaky }
    c: char;                       { prave spracuvany znak }
begin
   while read(c) do begin          { citame zo vstupu, kym sa da }
      if c='a' then push(a, c);    { znak vlozime do spravneho zasobniku }
      if c='b' then push(b, c);
   end;
   while not empty(a) and not empty(b) do begin
      pop(a); pop(b);              { vyberame znaky z oboch zasobnikov }
   end;
   if empty(a) and empty(b) then write('1')  { su prazdne oba? }
   else write('0');
end;
```

Tento algoritmus má lineárnu časovú aj pamäťovú zložitosť a potrebuje dva zásobníky.

Riešenie 1b:

Na riešenie tejto úlohy stačilo použiť len jeden zásobník. Budeme si v ňom pamätať, o koľko viac znakov `a` ako znakov `b` sme doteraz prečítali. Presnejšie, ak bol tento rozdiel kladný, budeme mať v zásobníku príslušný počet znakov `+`, inak tam budeme mať príslušný počet znakov `-`.

Všimnime si napríklad, čo sa stane, keď zo vstupu prečítame ďalšie `a`. Rozdiel sa zvýšil o 1, potrebujeme teda upraviť zásobník. Skontrolujeme, aký znak je na jeho vrchu. Ak je to `-`, len ho odstránime. Ak je tam `+` alebo je zásobník prázdny, pridáme nové `+`. Znak `b` spracujeme opačne.

```
{ Pomocna funkcia, ktora zisti znak na vrchu zasobnika }
function peek(var s:stack of char): char;
var c: char;
begin
   if empty(s) then c := '0'       { ak je prazdny, vratime napr. nulu }
   else begin
      c := pop(s);                 { inak odoberieme prvok zo zasobnika }
```

```
      push(s, c);                  { vlozime ho hned spat }
   end;
   peek := c;                      { a odovzdame ho ako navratovu hodnotu }
end;

var r: stack of char;              { tu je ulozeny rozdiel a-b }
    c: char;                       { prave spracuvany znak }
begin
   while read(c) do begin
      if c='a' then                              { zvysujeme rozdiel }
         if peek(r)='-' then pop(r) else push(r, '+');
      if c='b' then                              { znizujeme rozdiel }
         if peek(r)='+' then pop(r) else push(r, '-');
   end;
   if empty(r) then write('1') else write('0'); { je rozdiel nulovy? }
end;
```

Na spracovanie každého znaku nám stačí konštantný počet príkazov, preto je časová zložitosť naďalej lineárna. V zásobníku bude najviac toľko znamienok, koľko je znakov na vstupe, preto je aj pamäťová zložitosť lineárna.

Zadania domáceho kola kategórie B

B-I-1 Diaľnica

Píše sa rok 2108. Bratislavu a Košice konečne spojila diaľnica. Keď to Jano, bývajúci v Košiciach, uvidel, od radosti sa hneď rozbehol kúpiť si auto.

Rýchlo však zistil, že auto nejazdí na dobré slovo, ale treba doň občas natankovať benzín. Keď natankuje plnú nádrž, dokáže na svojom aute prejsť po diaľnici až K metrov.

Pozdĺž diaľnice sa nachádza N čerpacích staníc, príčom i-ta stanica sa nachádza a_i metrov od Košíc. Jana zaujímajú také dvojice staníc, ktorých vzdialenosť je menšia ako dojazd jeho auta – teda ak na jednej z nich natankuje, dokáže sa potom dostať na druhú z nich.

Súťažná úloha:

Napíšte program, ktorý načíta hodnoty K, N a a_i a spočíta počet dvojíc čerpacích staníc, ktorých vzdialenosť je menšia alebo rovná K.

Formát vstupu:

Prvý riadok vstupu obsahuje jedno celé číslo K ($1 \leq K \leq 1\,000\,000\,000$) – maximálnu vzdialenosť, ktorú prejde Janovo auto na jedno natankovanie.

Druhý riadok obsahuje počet čerpacích staníc N ($1 \leq N \leq 200\,000$, v polovici testovacích vstupov bude dokonca platiť $1 \leq N \leq 2\,000$).

Tretí riadok obsahuje N medzerami oddelených celých čísel $a_1, \ldots, a_N$ ($0 \leq a_1 < a_2 < \cdots < a_N \leq 1\,000\,000\,000$) – vzdialenosti jednotlivých čerpacích staníc od Košíc.

Formát výstupu:

Výstup má obsahovať jediný riadok a v ňom jediné celé číslo P – počet dvojíc čerpacích staníc, ktorých vzdialenosť je najviac K.

Môžete predpokladať, že testovacie vstupy budú zvolené tak, aby hodnota P nikdy neprekročila 2 miliardy.

Príklady:

Vstup	Výstup
100 3 15 80 120	2

Prvá a tretia stanica sú vo vzdialenosti 105, čo je priveľa.

Vstup

```
100
5
15 65 80 100 115
```

Výstup

```
10
```

Každé dve stanice sú vo vzdialenosti 100 alebo menej.

Odovzdávanie riešení:
Toto je praktická úloha. Odovzdávate **len funkčný, odladený program** prostredníctvom webového rozhrania.

B-I-2 Kolotoč

V lunaparku majú nový kolotoč. Je unikátny tým, že ide o jediný štvorcový kolotoč na celom svete. Presnejšie, tvorí ho N^2 sedadiel umiestnených do štvorca veľkosti $N \times N$.

Keď je kolotoč pustený, každú sekundu sa sedadlá presunú na nové pozície. Jedno otočenie vyzerá nasledovne: Sedadlá, ktoré sú na obvode kolotoča, sa posunú o 1 v smere hodinových ručičiek. Keď si tieto sedadlá odmyslíme, dostaneme kolotoč $(N-2) \times (N-2)$, ktorý otočíme rovnakým spôsobom. (Pre nepárne N nám v strede ostane jedno sedadlo, ktoré sa točí na mieste.)

Súťažná úloha:
Daná je hodnota N, mená detí sediacich na kolotoči, a kladné číslo K. Napíšte program, ktorý načíta tieto údaje a vypíše, ako bude kolotoč vyzerať po K sekundách točenia sa.

Formát vstupu:
Prvý riadok vstupu obsahuje jedno celé číslo N ($1 \leq N \leq 1\,000$) – rozmer strany kolotoča.

Nasleduje N riadkov, každý obsahuje N mien detí. Mená detí sú tvorené veľkými a malými písmenami a majú dĺžku 1 až 6 znakov. Medzi každými dvoma menami je jedna medzera. Na kolotoči môže sedieť viac detí s rovnakými menami.

V poslednom riadku je jedno celé číslo K ($1 \leq K \leq 1\,000\,000$) – počet otočení kolotoča.

V polovici testovacích vstupov bude navyše platiť $N, K \leq 100$.

Formát výstupu:

Výstup má obsahovať N riadkov, v ktorých bude popísaný záverečný stav kolotoča. Všetky mená vypíšte zarovnané doprava na 7 znakov. (Dĺžka každého riadku výstupu teda musí byť presne $7N$.) Kratšie mená doplňte zľava medzerami.

Rada: v jazyku C môžete na správne formátovaný výpis premennej `meno` použiť príkaz `printf("%7s",meno);` a v jazyku Pascal príkaz `write(meno:7);`.

Príklad:

Vstup

```
4
Alenka Julka Monika Danka
Lucka Misko Mirko Petko
Jozko Milan Risko David
Jano Katka Misko Martin
2
```

Výstup

```
  Jozko  Lucka Alenka  Julka
   Jano  Risko  Milan Monika
  Katka  Mirko  Misko  Danka
  Misko Martin  David  Petko
```

Odovzdávanie riešení:

Toto je praktická úloha. Odovzdávate **len funkčný, odladený program** prostredníctvom webového rozhrania.

B-I-3 Balíčky

Mnohé distribúcie operačných systémov poskytujú svojim používateľom softvér rozdelený do balíčkov. Používateľ si prezrie zoznam ponúkaných balíčkov, vyberie si tie, ktoré chce používať, a operačný systém mu automaticky nainštaluje všetko potrebné.

Z pohľadu používateľa je to jednoduché a príjemné. O niečo ťažšie to má operačný systém. Niektoré balíčky totiž na sebe závisia. Presnejšie, môže sa stať, že na to, aby balíček i fungoval správne, treba mať nainštalované aj balíčky j a k.

V tejto úlohe sa budeme zaoberať zjednodušenou verziou takéhoto systému balíčkov.

Súťažná úloha:

Používateľ má k dispozícii N rôznych balíčkov, ktoré ešte nemá nainštalované. Pre každý balíček i máme danú jeho veľkosť v_i a zoznam iných balíčkov,

na ktorých tento balíček závisí. (Ak balíček i závisí na balíčkoch j a k, tak ak chceme nainštalovať balíček i, musíme nainštalovať aj j, aj k.)

Používateľ si vybral M balíčkov, ktoré potrebuje. Zistite celkovú veľkosť balíčkov, ktoré treba nainštalovať.

Formát vstupu:

Prvý riadok vstupu obsahuje jedno celé číslo N ($1 \leq N \leq 10\,000$) – počet balíčkov. Balíčky majú priradené čísla od 1 po N.

Nasledujúcich N riadkov popisuje jednotlivé balíčky. Presnejšie, i-ty riadok začína názvom balíčka i a jeho veľkosťou v_i ($1 \leq v_i \leq 10\,000$) v kilobajtoch. Nasleduje počet p_i iných balíčkov, na ktorých balíček i závisí. Zvyšok riadku tvorí p_i navzájom rôznych čísel týchto balíčkov.

Môžete predpokladať, že $p_1 + \cdots + p_N \leq 100\,000$ a že v závislostiach medzi balíčkami nie sú cykly. (Nemôže sa teda stať že by napr. balíček i závisel na balíčku j, ten na balíčku k, a ten na balíčku i.)

Názvy balíčkov neobsahujú medzeru a majú najviac 20 znakov. Medzi každou dvojicou údajov v riadku je práve jedna medzera.

V predposlednom riadku vstupu je jedno celé číslo M ($0 \leq M \leq N$) – počet balíčkov, ktoré chce používateľ.

V poslednom riadku je M rôznych čísel (oddelených medzerami), popisujúcich tieto balíčky.

Formát výstupu:

Výstup má obsahovať jediný riadok a v ňom celkovú veľkosť balíčkov v kilobajtoch, ktoré treba nainštalovať.

Príklad:

Vstup

```
6
python 300 1 6
bittorrent 500 2 1 3
ncurses 550 1 6
gdb 4700 0
freepascal 1200 2 4 6
glibc 100 0
1
2
```

Výstup

```
1450
```

Používateľ chce balíček `bittorrent`. *Tento závisí na balíčkoch* `python` *a* `ncurses`, *ktoré oba závisia na balíčku* `glibc`.
Všimnite si, že balíček `glibc` *stačí nainštalovať raz. Ostatné dva balíčky nainštalovať netreba.*

Odovzdávanie riešení:

Toto je teoretická úloha. Riešenie, spĺňajúce požiadavky uvedené v pravidlách, buď pošlite poštou na adresu vašej krajskej komisie, alebo odovzdajte vo formáte PDF prostredníctvom webového rozhrania.

B-I-4 Divné jazyky

Kdesi v oceáne medzi Polynéziou a Patagóniou objavili nedávno moreplavci časom zabudnuté súostrovie, ktoré nazvali Polygónia. Na mnohých ostrovoch tohto súostrovia žijú domorodé kmene, ktoré používajú vskutku podivné jazyky. Moreplavci zistili, že tieto kmene poznajú len dve hlásky: `U` a `A`. Z týchto hlások sú zložené všetky ich slová.

Súťažná úloha:

a) (1 bod)
V kmeni Krivozubých používajú jazyk, ktorý má nasledujúce vlastnosti:
– Žiadne slovo nie je dlhšie ako 8 znakov.
– Každé slovo znie odpredu rovnako ako odzadu.

Teda napríklad môžu používať slová `A`, `AA` a `UAUAU`, ale nie `UUAA` (lebo odzadu je to `AAUU`, čo nie je to isté ako `UUAA`) ani `AAAUUUAAA` (lebo je pridlhé).

Koľko najviac slov môže mať ich jazyk?

b) (2 body)
Aj v kmeni Dlhonosých používajú jazyk, kde každé slovo znie odpredu rovnako ako odzadu. Avšak Dlhonosí majú väčšiu slovnú zásobu, ich slová môžu mať nanajvýš 47 znakov. Koľko najviac slov môže mať ich jazyk?

c) (3 body)
Kmeň Strapatých má naozaj podivuhodný jazyk. Všetky slová v ich jazyku majú presne 7 hlások. Navyše platí, že ak sa domorodec z tohto kmeňa pomýli v ľubovoľnej jednej hláske ľubovoľného slova, aj tak sa dá spoznať, ktoré slovo chcel povedať.

V jazyku tohto kmeňa teda napríklad nemôže byť dvojica slov `UUUAAAU` a `UUUAAUA` – keby niekto povedal `UUUAAAA`, nebolo by jasné, ktoré z týchto dvoch slov chcel povedať. Z rovnakého dôvodu môže byť v tomto jazyku najviac jedno zo slov `UUUAAAU` a `UUUAAAA`.

Nájdite jazyk, ktorý spĺňa tieto požiadavky a má čo najviac slov. (Ak chcete, môžete si samozrejme pri hľadaní pomôcť počítačom, nezabudnite však v takom prípade v riešení uviesť aj použitý program.)

d) (4 body)
Dokážte, že jazyk kmeňa Strapatých nemôže mať viac ako 16 slov.

Odovzdávanie riešení:

Toto je teoretická úloha. Riešenie, spĺňajúce požiadavky uvedené v pravidlách, buď pošlite poštou na adresu vašej krajskej komisie, alebo odovzdajte vo formáte PDF prostredníctvom webového rozhrania.

Zadania krajského kola kategórie A

A-II-1 Horár Jedlička

Pán Jedlička kedysi dávno vysadil krásny a veľký les, stromy rástli v radách rozmiestnené s dokonalou presnosťou. Nastal čas, keď les vyrástol a začalo sa s ťažbou dreva. Každý drevorubač začal klátiť stromy na inom okraji lesa, a vysekal do neho poriadnu paseku. Keď sa pán Jedlička prišiel pozrieť na les, objavil už len jeho zvyšok, s hlbokými výsekmi od ťažby. Pána Jedličku by teraz zaujímalo, koľko stromov mu vlastne ostalo. Pomôžete mu s tým?

Súťažná úloha:

Stromy v lese pána Jedličku sú zasadené tak, že rastú v mrežových bodoch obdĺžnikovej mriežky. Polohu každého stromu môžeme teda popísať dvojicou celočíselných súradníc (x, y). To, čo z lesa zostalo, je vymedzené mnohouholníkom (ktorý môže byť aj nekonvexný) tak, že vrcholy mnohouholníka sú stromy, a na všetkých mrežových bodoch vnútri alebo na hranici mnohouholníka stále ešte stojí strom. Vašou úlohou je spočitať, koľko mrežových bodov leží vnútri tohoto mnohouholníka, vrátane jeho hranice.

Formát vstupu:

Prvý riadok obsahuje prirodzené číslo N – počet vrcholov mnohouholníka ($3 \leq N \leq 100\,000$).

Na nasledujúcich N riadkoch sú popísané jednotlivé vrcholy mnohouholníka. Na i-tom z nich sa nachádza dvojica celočíselných súradníc $1 \leq x_i \leq 10^9$, $1 \leq y_i \leq 10^9$ udávajúca polohu i-teho vrcholu. Vrcholy sú zadané v poradí, v akom ležia na hranici mnohouholníka pri obchádzaní v smere alebo proti smeru hodinových ručičiek.

Formát výstupu:

Na jediný riadok výstupu vypíšte jedno celé číslo predstavujúce počet mrežových bodov vnútri mnohouholníka vrátane jeho hranice.

Príklady:

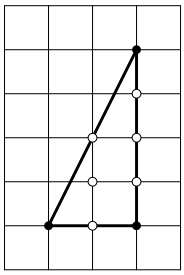
Vstup

```
3
1 1
3 5
3 1
```

Výstup

```
9
```

Tri mrežové body tvoria vrcholy trojuholníka, päť mrežových bodov leží na hranách a jeden mrežový bod leží vo vnútri.



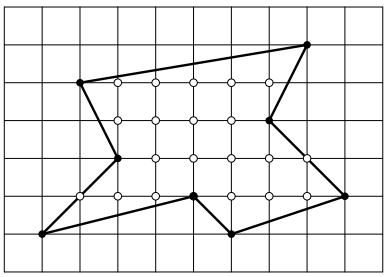
Vstup

```
8
2 5
3 3
1 1
5 2
6 1
9 2
7 4
8 6
```

Výstup

```
28
```

Osem mrežových bodov tvoria vrcholy mnohouholníka, dva mrežové body ležia na hranách a osemnásť mrežových bodov leží vo vnútri.



A-II-2 Babylonská kríza

V Babylone žila žena, ktorá vraj rozprávala všetkými jazykmi sveta. Vyučovala všetkých Babylončanov aby sláva Babylonu neupadla. Keď sa na svete objavil nový jazyk, tak ho táto žena, tiež Enochiou zvaná, znenazdania vedela tiež. Avšak ako šli roky, tak Enochia starla a čím ďalej tým viac sa bála toho, že by mohla začať jazyky zabúdať – a čo by sa potom stalo s jej Babylonom? Určite by aj veža padla z toľkého nešťastia.

Tak sa teda Enochia rozhodla, že by bolo načase spoľahnúť sa na dáku „techniku“. Sadla si, zobrala dláto a kladivko, a ku každému jazyku čo vedela spísala *wordlist* – zoznam všetkých jeho slov. Potom zvolala vynálezcov, alchymistov, najlepších z najlepších, ale nik si nevedel rady, ako zrealizovať nápad, ktorý mala Enochia. Chcela totiž, aby vynašli spôsob, či snáď prístroj, ktorý by vedel zobrať ľubovoľný text a odhaliť, akýmže to jazykom je písaný.

Súťažná úloha:

Máte k dispozícii pomocný súbor `wordlisty.in`, v ktorom je pre každý známy jazyk uložený zoznam jeho slov. Presný formát tohto súboru je popísaný nižšie.

Vašou úlohou je navrhnúť program, ktorý po spustení načíta obsah pomocného súboru a užívateľom zadaný text a určí jazyk, v ktorom je daný text napísaný.

Presnejšie, vašou úlohou je nájsť ten jazyk, do ktorého patrí najviac slov zadaného textu. (Ak sa nejaké slovo vyskytne v texte viackrát, počíta sa samostatne každý jeho výskyt.)

Formát súboru wordlisty.in:

V prvom riadku je číslo N ($1 \leq N \leq 10\,000$), ktoré určuje počet wordlistov v súbore. Ďalej nasleduje N wordlistov.

Každý z wordlistov začína dvomi riadkami. V prvom je meno jazyka a v druhom číslo S, ktoré udáva počet slov v tomto wordliste. Potom nasleduje S riadkov, pričom v každom z nich je práve jedno slovo daného jazyka. (Každé slovo je kratšie než 255 znakov a je zložené len z malých písmen anglickej abecedy. Slová v jazyku sú navzájom rôzne a nie sú uvedené v žiadnom konkrétnom poradí.)

Celkový počet slov vo všetkých wordlistoch je najviac 100 000. Meno každého jazyka je tvorené najviac 255 malými písmenami anglickej abecedy. Môžete predpokladať, že mená jazykov sú navzájom rôzne. Rôzne wordlisty môžu obsahovať rovnaké slová.

Formát vstupu:

Na štandardnom vstupe je text, ktorý chce Enochia zanalyzovať. Text sa skladá zo slov. Opäť, každé slovo je zložené len z malých písmen anglickej abecedy a má najviac 255 znakov. Slová sú oddelené medzerami alebo koncami riadkov. Text nebude obsahovať viac než 1 000 000 slov. Slová v texte nemusia byť navzájom rôzne.

Formát výstupu:

Vypíšte (v ľubovoľnom poradí) všetky jazyky, v ktorých sa nachádza najviac slov zadaného textu.

Príklad:

wordlisty.in

```
3
slovencina
4
text
bager
ahoj
zebra
anglictina
4
zebra
by
hello
car
nemcina
3
hallo
auto
sagt
```

vstup

```
tento text
by nepochopila
ani zebra
co ma zebra auto
```

výstup

```
slovencina
anglictina
```

(Aj v slovenčine, aj v angličtine sa vyskytujú po tri slová textu, v nemčine len jedno.)

A-II-3 Cyklistické preteky

Po úspechu horského maratónu sa na vás jeho organizátori obrátili s prosbou o pomocou pri organizovaní cyklistických pretekov. Trasa pretekov by mala viesť z cieľa horského maratónu, Výšných Hákov, do hlavného mesta, Veľkého Sumca. Cyklistické preteky budú tvorené niekoľkými etapami a organizátori už určili možné dvojice miest, medzi ktorými by mohli viesť jednotlivé etapy pretekov. Pre každú takúto dvojicu naviac odhadli počet divákov, ktorí by sa prišli pozrieť na preteky. Pretože rozpočet celých pretekov je obmedzený, organizátori by radi trasu pretekov navrhli tak, aby mala čo najmenej etáp, a pritom by ich videlo čo najviac divákov.

Súťažná úloha: Váš program dostane zoznam dvojíc miest, medzi ktorými by mohli viesť etapy pretekov, a pre každú dvojicu odhad počtu divákov, ktorí by sa na danú etapu prišli pozrieť. Jednotlivé etapy, ktoré tvoria trasu pretekov, musia na seba v koncových mestách nadväzovať. Program by mal nájsť trasu pretekov vedúcu z Vyšných Hákov do Veľkého Sumca, ktorá má čo najmenej etáp a medzi všetkými takými trasami určiť takú, ktorú uvidí čo najväčší počet divákov (počty divákov jednotlivých etáp sa sčítavajú). Pokiaľ je takých trás viac, program môže vypísať ľubovoľnú z nich.

Prvý riadok vstupu obsahuje dve prirodzené čísla N a M, $2 \leq N \leq 100\,000$ a $1 \leq M \leq 1\,000\,000$; N je počet miest a M je počet dvojíc miest, medzi ktorými by mohla viesť jedna etapa pretekov. Mestá sú očíslované od 1 po N, pričom Vyšné Háky majú číslo 1 a Veľký Sumec má 2. Na každom z M nasledujúcich riadkov je trojica čísel A, B a D ($1 \leq A, B \leq N$ a $0 \leq D \leq 1\,000$) popisujúca jednu z dvojice miest, medzi ktorými by mohla viesť jedna etapa: etapa by mohla viesť medzi mestami s číslami A a B a očakávaný počet divákov pre túto etapu je D. Etapa pretekov môže ísť z mesta A do B alebo aj opačným smerom.

Môžete predpokladať, že existuje aspoň jedna možná trasa pretekov. Vstup naviac neobsahuje dva rôzne riadky, ktoré by opisovali etapu medzi rovnakou dvojicou miest.

Na prvý riadok výstupu vypíšte dve čísla: najmenší počet etáp P pretekov z 1 do 2 a najväčší počet divákov, ktorý by mohli pozerať takéto preteky. Na druhý riadok vypíšte $P + 1$ čísel miest, ktoré tvoria optimálnu trasu. Ak je optimálnych trás, vyberte si ľubovoľnú jednu z nich.

Príklad:

Vstup

```
6 9
1 6 10
1 3 8
4 6 2
4 3 7
5 6 0
5 3 4
4 5 100
2 4 1
2 5 2
```

Výstup

```
3 16
1 3 4 2
```

A-II-4 Zásobníkové počítače

Študijný text nájdete za zadaniami súťažnej úlohy A-I-4 z domáceho kola.

Súťažná úloha:

Napíšte program pre zásobníkový počítač, ktorý vyhodnotí zadaný logický výraz a vypíše jeho hodnotu na výstup. Program by mal používať čo najmenší počet zásobníkov.

Logické výrazy zapisujeme pomocou znakov `0`, `1`, `&`, `|`, `(` a `)`. Znaky `0` a `1` sú logické konštanty (`0` je nepravda, `1` je pravda), `&` je logický súčin (spojka a: `0&0=0&1=1&0=0`, `1&1=1`), `|` je logický súčet (spojka alebo: `0|0=0`, `0|1=1|0=1|1=1`) a zátvorky fungujú bežným spôsobom. Ak nie sú uvedené zátvorky, tak súčin má prednosť pred súčtom, a teda `1|0&0|1 = 1|(0&0)|0 = 1|0|0 = 1`.

Zadania krajského kola kategórie B

B-II-1 Binárny súčet

Bitlandia je jediná známa krajina, v ktorej obyvatelia poznajú len dvojkovú sústavu. Binárne čísla sa tam používajú úplne všade – v obchodoch, školách, na hotelových izbách, ba dokonca aj čísla električiek sú písané v tejto sústave. Obyvatelia Bitlandie takisto radi riešia rôzne krížovky či hlavolamy. Najnovším hitom sa stal takzvaný Bitlandský súčtový hlavolam. Zadanie hlavolamu pozostáva z troch binárnych čísel (označme ich A, B, C), pričom niektoré cifry v číslach A, B chýbajú. Úlohou riešiteľa je doplniť chýbajúce cifry tak, aby platil súčet $A + B = C$. Nasledujúci obrázok ilustruje jedno zadanie hlavolamu.

```
 0010**0
 001**1*
---------
 0101001
```

Jedno z možných riešení tohoto hlavolamu je súčet $0010110 + 0010011 = 0101001$.

Súťažná úloha:

Napíšte program, ktorý načíta reťazce A, B, C a rozhodne, či zadaný hlavolam má riešenie. Ak hlavolam je riešiteľný, váš program by mal vypísať aj jedno riešenie.

Formát vstupu a výstupu:

Vstup obsahuje tri riadky, ktoré postupne obsahujú reťazce A, B a C. Chýbajúce cifry sa vyskytujú len v reťazcoch A a B a sú označené znakom `'*'`. Môžete predpokladať, že všetky tri reťazce na vstupe majú rovnakú dĺžku.

V prípade, že hlavolam zo vstupu má riešenie, váš program by mal vypísať jedno riešenie – dva riadky, určujúce sčítance hlavolamu v binárnom zápise. Ak hlavolam nemá riešenie, výstupom by mal byť jediný riadok s textom „`Nema riesenie.`“.

Príklady:

Vstup	Výstup

`010**0`
`01**1*`
`101001`
	`010110`
	`010011`

Vstup	Výstup
01** 01*0 1111	Nema riesenie.

Hodnotenie:

- Riešenia, ktoré dokážu efektívne spracovať 100 000-ciferné reťazce, môžu získať maximálne 10 bodov.
- Riešenia, ktoré dokážu efektívne spracovať 1 000-ciferné reťazce, môžu získať maximálne 8 bodov.
- Riešenia, ktoré dokážu efektívne spracovať 10-ciferné reťazce, môžu získať maximálne 5 bodov.

B-II-2 Výber dovolenky

Peter sa rozhodol, že si už teraz v zime naplánuje dovolenku. Chce ísť do jedného letoviska, kde sa každý večer koná jedna atrakcia. Petra konkrétne zaujímajú koncerty, futbalové zápasy a preteky členov v prístave. Chcel by počas svojej dovolenky každú z týchto atrakcií zažiť aspoň raz. Dovolenka samozrejme musí byť jedno súvislé časové obdobie, a keďže Peter chce za ňu zaplatiť čo najmenej, čím bude kratšia, tým lepšie. Pomôžte Petrovi nájsť najlepšie obdobie na dovolenku.

Súťažná úloha:

Daný je reťazec tvorený N malými písmenami anglickej abecedy. Tento reťazec popisuje atrakcie, ktoré sa budú v jednotlivé noci v letovisku konať. Písmeno `a` predstavuje koncert, písmeno `b` futbalový zápas, `c` sú preteky členov a ostatné písmená predstavujú iné atrakcie. Napíšte program, ktorý nájde dĺžku najkratšieho úseku, ktorý aspoň raz obsahuje každú z troch udalostí, ktoré Petra zaujímajú.

Formát vstupu a výstupu:

V prvom riadku je jedno celé číslo N udávajúce počet dní, z ktorých si Peter môže vyberať. Druhý riadok vstupu obsahuje N-znakový reťazec popisujúci atrakcie.

Vypíšte jeden riadok a v ňom jedno celé číslo – najmenší počet dní, ktoré musí Peter stráviť na dovolenke.

Príklad:

Vstup	Výstup
10 abaacazaba	4

Od 2. do 5. dňa zažije všetky tri udalosti. Žiadny kratší úsek nevyhovuje.

Hodnotenie:

- Riešenia fungujúce pre $N \leq 10\,000\,000$ môžu získať maximálne 10 bodov.
- Riešenia fungujúce pre $N \leq 1\,000\,000$ môžu získať maximálne 9 bodov.
- Riešenia fungujúce pre $N \leq 5\,000$ môžu získať maximálne 7 bodov.
- Riešenia fungujúce pre $N \leq 500$ môžu získať maximálne 5 bodov.

B-II-3 Sánkovanie

Malý Kleofáš našiel pod stromčekom nové sánky. A odvtedy vyzerajú všetky jeho dni rovnako – akonáhle môže, vytiahne sánky na kopec za domom, spustí sa dole, zase vytiahne sánky na kopec... a tak dokola, až kým ho mama nezavolá domov.

Po čase však aj Kleofáša omrzelo spúšťať sa len tak, a tak si začal plánovať rôzne trasy: Z vrcholu sa napríklad pustí k osamelej jabloni, od nej k odstavenému traktoru, odtiaľ k mostíku, a od mostíka rovno dole.

Kleofáš si na papieri nakreslil schému kopca. Na nej vyznačil N zaujímavých miest a očísloval ich od 1 do N. Miesto číslo 1 je vrchol kopca, kde Kleofáš každú jazdu začína. Miesto číslo N je úpätie kopca, kde každú jazdu končí. (Pozor, ostatné miesta **nemusia byť** očíslované podľa nadmorskej výšky!)

Medzi niektorými dvojicami zaujímavých miest sa dá priamo pustiť na saniach, medzi inými nie. Kleofáš si vypísal všetkých M dvojíc miest, medzi ktorými sa pustiť dá. Teraz sedí nad svojim papierom a snaží sa porátať, koľkými rôznymi cestami sa vlastne môže z kopca spustiť.

Súťažná úloha:

Napíšte program, ktorý načíta údaje z Kleofášovho papiera a spočíta hľadaný počet ciest.

Formát vstupu a výstupu:

Prvý riadok vstupu obsahuje celé číslo N ($2 \leq N \leq 100\,000$), udávajúce počet zaujímavých miest. Druhý riadok vstupu obsahuje celé číslo M ($1 \leq M \leq 1\,000\,000$), udávajúce počet dvojíc miest, medzi ktorými sa dá priamo spustiť. Nasleduje M riadkov, v každom z nich sú dve čísla a a b hovoriace, že z miesta a sa dá priamo spustiť na miesto b.

Popis vstupu skutočne zodpovedá miestam na kopci. Ak sa vieme spustiť z a do b, znamená to, že a má väčšiu nadmorskú výšku ako b.

Pri písaní programu môžete predpokladať, že sa hľadaný počet ciest zmestí do bežnej celočíselnej premennej.

Vypíšte jeden riadok a v ňom jediné celé číslo, udávajúce počet spôsobov, ako sa dá spustiť z kopca popísaného na vstupe.

Príklad:

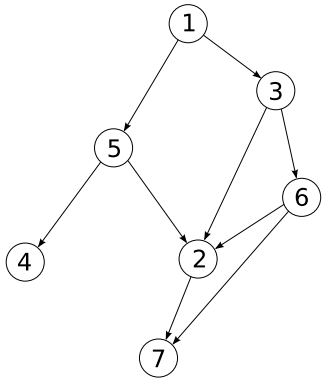
Vstup

```
7
9
1 5
3 2
5 2
5 4
1 3
3 6
6 7
6 2
2 7
```

Výstup

```
4
```

Takto vyzerá zadaný kopec:



Možné cesty vyzerajú nasledovne:
$1 \to 5 \to 2 \to 7$, $1 \to 3 \to 2 \to 7$, $1 \to 3 \to 6 \to 2 \to 7$ a $1 \to 3 \to 6 \to 7$.

Hodnotenie:

- Riešenie zvládajúce limity uvedené pri formáte vstupu môže získať 10 bodov.
- Nanajvýš 8 bodov dostanete za algoritmus, ktorý na bežnom počítači do sekundy skončí pre $N \leq 2\,000$.
- Nanajvýš 6 bodov dostanete za algoritmus, ktorý vyrieši pôvodnú úlohu za predpokladu, že zaujímavé miesta sú očíslované podľa nadmorskej výšky. (Teda že pre každú dvojicu „a b“ na vstupe platí $a < b$.)
- Nanajvýš 5 bodov dostanete za algoritmus, ktorý vyrieši túto zjednodušenú úlohu pre $N \leq 2\,000$.
- Aspoň 3 body môžete získať za ľubovoľné funkčné riešenie.

B-II-4 Banka

V mestečku Kocúrkovo onedlho otvoria novú banku. Čísla účtov v tejto banke budú mať práve N cifier. Číslo účtu môže začínať aj nulami, teda napríklad 0047 je platné číslo účtu pre $N = 4$.

Banka by chcela prideľovať čísla účtov zákazníkom tak, aby ich ochránila pred následkom preklepu pri zadávaní čísla účtu. Presnejšie, ak napíšeme ľubovoľné číslo existujúceho účtu a pomýlime sa pri tom v jednej cifre, nesmieme dostať číslo iného existujúceho účtu.

a) **(1 bod)** Ukážte, že pre $N = 2$ môže mať Kocúrkovská banka 10 zákazníkov.

b) **(2 body)** Dokážte, že pre žiadne N nemôže mať Kocúrkovská banka viac ako 10^{N-1} zákazníkov.

c) **(4 body)** Navrhnite, ako má Kocúrkovská banka vyberať čísla účtov, aby mohla mať čo najviac zákazníkov. (Plný počet bodov za túto podúlohu dostanete, ak sa vám podarí nájsť spôsob ako zostrojiť 10^{N-1} vyhovujúcich čísel účtov.)

d) **(3 body)** Aj v susedných dvoch dedinách, Popletenej Vieske a Bezpečnej Trieske, otvárajú nové banky, a tie majú ešte lepšie ponuky.

Banka v Popletenej Vieske bude voliť čísla účtov tak, že keď zákazník napíše svoje číslo účtu a pomýli sa **v nanajvýš 4 cifrách**, nikdy nedostane číslo iného existujúceho účtu.

Banka v Bezpečnej Trieske bude voliť čísla účtov tak, že keď zákazník napíše svoje číslo účtu a pomýli sa **v nanajvýš 2 cifrách**, nielenže určite nedostane číslo iného existujúceho účtu, ale navyše bude banka vždy vedieť jednoznačne určiť jeho skutočné číslo účtu.

Ktorá banka môže mať viac zákazníkov, ak obe používajú 10-ciferné čísla účtov a volia ich najlepšie ako sa dá?

(Čísla účtov v rôznych bankách sú navzájom nezávislé, môže sa stať, že nejaký zákazník prvej banky bude mať trebárs aj rovnaké číslo účtu ako iný zákazník druhej banky.)

Zadania celoštátneho kola kategórie A

A-III-1 Horár Jedlička II

V krajskom kole pán Jedlička s vašou pomocou a s hrôzou zistil, že z jeho kedysi krásneho a veľkého lesa zostáva len drobný háj. Aby zabránil jeho ďalšiemu zmenšovaniu, najal si tlupu strážnych trollov. Trollovia sú známi svojou silou, menej však už svojou prenikavou inteligenciou. Pán Jedlička sa rozhodol príkazy pre trollov čo najviac zjednodušiť. Každý troll teda dostal pridelené 2 body a má chodiť po úsečke, ktorá ich spája, tam a späť.

Bohužial počas prvého dňa bolo nutné ošetriť skoro všetkých trollov kvôli drobným zraneniam. Pán Jedlička zabudol, že sa úsečky, po ktorých trollovia pochodujú, môžu pretínať, a vo svojích inštrukciách zabudol uviesť, aby sa trollovia vyhýbali zrážkam s ostatnými trollmi. Pokus pridať trollom tento príkaz narazil na ich zábudlivosť. Po niekoľkých opakovaniach ťažkého príkazu „na konci úsečky sa otoč“ trollom pretiekol zásobník a príkaz „vyhýbaj sa zrážkam“ sa stratil, čo malo nepriaznivé dôsledky na rozpočet ošetrovne.

Pán Jedlička sa preto rozhodol všetky križovatky označiť dopravnou značkou „Pozor troll!“. Vašou úlohou je zistiť, kam má tieto značky umiestniť.

Súťažná úloha: Úsečka, po ktorej sa i-ty troll pohybuje, je zadaná dvojicou jej krajných bodov so súradnicami $(a_i, b_i), (c_i, d_i)$, kde a_i, b_i, c_i a d_i sú celé čísla. Krajné body úsečky neležia na žiadnej inej úsečke. Vašou úlohou je vypísať súradnice priesečníkov týchto úsečiek. Každý priesečník vypíšte iba raz, aj keby sa v ňom pretínali 3 a viac úsečiek. Predpokladajte, že priesečníkov je málo – podstatne menej ako N^2.

Formát vstupu: Prvý riadok obsahuje prirodzené číslo N, udávajúce počet trollov, $0 \leq N \leq 10\,000$. Na nasledujúcich N riadkoch je popis úsečiek, po ktorých sa trollovia pohybujú. Na i-tom riadku sa nachádza štvorica celých čísel a_i, b_i, c_i a d_i, súradnice krajných bodov (a_i, b_i) a (c_i, d_i) úsečky. Môžete predpokladať, že $(a_i, b_i) \neq (c_i, d_i)$.

Formát výstupu: Na každý riadok výstupu vypíšte súradnice jedného z priesečníkov zadaných úsečiek. Poradie priesečníkov môže byť ľubovoľné.

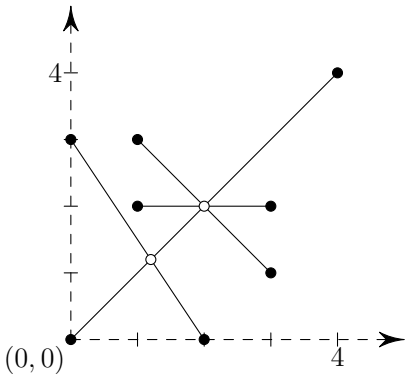
Príklad:

Vstup

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

Výstup

```
1.2 1.2
2 2
```



A-III-2 Magické cestovanie

V dôsledku finančnej krízy značne poklesol záujem o magické vystúpenia kúzelníka Davida Aluminumfielda. Preto sa starý kúzelník rozhodol, že čas, ktorý takto nadobudol, využije na precestovanie vzdialených kútov zeme. Klesajúce tržby majú ale za následok, že si nemôže dovoliť konvenčné spôsoby prepravy a musí sa spoľahnúť na osvedčené metódy svojich kúzelníckych predkov.

Začal teda pátrať na povale svojho obydlia a medzi veľkou hromadou zbytočných a starých vecí sa mu podarilo nájsť knihu cestovných kúzel. Jej použitie ale nie je úplne jednoduché. Funguje nasledovne: David si môže vybrať, kde v knihe začne čítať, musí však prečítať súvislý úsek textu – nesmie preskakovať slová ani sa vracať späť. Každé slovo v knihe je nabité nejakou magickou energiou. Zaklínadlo sa podarí len vtedy, ak je celková magická energia prečítaných slov násobkom magickej konštanty K. Zaklínadlo bude tým silnejšie, čím viac slov ho tvorí. (Na veľkosti celkovej magickej energie nezáleží.)

Súťažná úloha:

Daná je magická konštanta K, počet slov N v knihe kúzel, a pre každé slovo jedno celé číslo udávajúce jeho magickú energiu. Vašou úlohou bude na základe týchto informácii nájsť najvhodnejší začiatok a koniec zaklínadla – t. j. najdlhší súvislý úsek slov, pre ktorý platí, že súčet ich energických hodnôt je násobkom čísla K.

Číslo K je obyčajne oveľa menšie ako počet slov v knihe.

Formát vstupu:

V prvom riadku vstupu sú dve prirodzené čísla – N a K, $1 \leq N \leq 1\,000\,000$ a $1 \leq K \leq 50\,000$, oddelené medzerou. Ako bolo povedané, K je obvykle omnoho menšie ako N. V druhom riadku vstupu je medzerami oddelených N nezáporných čísel $a_1, \ldots, a_N, 0 \leq a_i \leq 1,000,000,000$, ktoré predstavujú magickú energiu jednotlivých slov.

Formát výstupu:

Program vypíše dve čísla – i a $j (1 \leq i \leq j \leq N)$, pričom súčet $a_i + a_{i+1} + \ldots + a_j$ musí byť násobkom čísla K a rozdiel $j - i$ najvyšší možný spomedzi všetkých takýchto dvojíc. Ak je možných dvojíc viac, vypíšte ľubovoľnú z nich. Naopak, ak neexistuje žiadna taká dvojica, vypíšte `Neda sa zaklinat`.

Príklad 1:

Vstup

```
8 5
1 2 8 6 3 4 4 9
```

Výstup

```
3 7
```

Takisto výstup „1 5“ by bol správny.

Príklad 2:

Vstup

```
5 8
1 1 1 1 1
```

Výstup

```
Neda sa zaklinat
```

A-III-3 Zásobníkové počítače

Študijný text nájdete za zadaniami súťažnej úlohy A-I-4 z domáceho kola.

Súťažná úloha:

Na vstupe je zadaný reťazec obsahujúci výhradne znaky `a`, `b`, `c` a `d`. Napíšte program pre zásobníkový počítač, ktorý zistí, či má každý zo znakov `a` až `d` rovnaký počet výskytov. Ak áno, nech váš program na výstup vypíše 1, ak nie, nech vypíše 0.

Zamerajte sa na požitie čo najmenšieho počtu zásobníkov aj za cenu vyššej časovej zložitosti.

Príklad: Na vstupy `abcddcba` a `aaabbbcccddd` je správna odpoveď 1.

Na vstup `badbadc` je správna odpoveď 0 (znaky `a`, `b` a `d` se vyskytujú dvakrát, ale znak `c` len raz).

A-III-4 Výlet do Švajčiarska

Tibor je vášnivý cyklista. Počas najbližších letných prázdnin by sa rád vybral bicyklovať do Švajčiarska. Ideálny výlet by sa podľa neho mal skladať zo štyroch jednodenných etáp. Po každej etape prespí v mestečku, kde skončila, a na druhý deň ráno odtiaľ vyrazí na ďalšiu etapu. Aby si mohol kúpiť spiatočný lístok na vlak do Švajčiarska (a tak ušetril), potreboval by začínať prvú etapu v tom istom meste ako končiť poslednú.

Tibor si na internete našiel zoznam všetkých možných jednodenných etáp po Švajčiarsku. Ku každej už dokonca iní cyklisti uviedli aj hodnotenie, aká je pekná.

Súťažná úloha:

Pre jednoduchosť očíslujeme mestá v Švajčiarsku číslami od 1 po N. Počet etáp, ktoré Tibor našiel, označíme M. Každá etapa spája dve rôzne mestá. Žiadne dve etapy nespájajú tú istú dvojicu miest. V každom meste končí nanajvýš 100 etáp.

Každú etapu je možné ísť oboma smermi. Každá etapa má číselné ohodnotenie, čo je celé číslo medzi 1 a 256. Ohodnotenie je rovnaké pre oba smery.

Vašou úlohou je nájsť postupnosť štyroch **rôznych** etáp, ktoré budú na seba nadväzovať, a navyše posledná bude končiť v meste, kde prvá začína. Ak je viac možností, vyberte tú, v ktorej je súčet ohodnotení etáp najväčší. Ak je ešte stále viac možností, vyberte ľubovoľnú z nich.

Formát vstupu:

V prvom riadku vstupu sú dve medzerou oddelené celé čísla N a M ($4 \leq N \leq 10\,000$ a $4 \leq M \leq 1\,000\,000$).

Nasleduje M riadkov, každý z nich popisuje jednu etapu. Presnejšie, v i-tom riadku sú tri medzerami oddelené prirodzené čísla x_i, y_i, h_i ($1 \leq x_i, y_i \leq N$, $x_i \neq y_i$ a $1 \leq h_i \leq 256$), kde x_i a y_i sú čísla miest, ktoré etapa spája, a h_i je jej ohodnotenie.

Môžete predpokladať obmedzenia uvedené v časti „Súťažná úloha“: Žiadne dve etapy nespájajú tú istú dvojicu miest a v každom meste končí nanajvýš 100 etáp.

Formát výstupu:

Ak žiadna vyhovujúca trasa neexistuje, vypíšte jeden riadok a v ňom reťazec „`NEEXISTUJE"` (bez úvodzoviek).

V opačnom prípade vypíšte dva riadky. V prvom z nich uveďte najväčší možný súčet ohodnotení etáp. V druhom vypíšte 5 čísel miest v poradí, v akom ich Tibor má navštíviť. (Prvé mesto musí byť rovnaké ako piate.)

Príklad 1:

Vstup

```
6 9
1 2 10
2 5 11
3 1 10
6 3 7
1 4 3
2 6 15
5 3 10
4 5 5
4 6 9
```

Výstup

```
43
2 6 3 5 2
```

Príklad 2:

Vstup

```
7 9
1 2 1
2 3 1
1 3 1
3 4 1
3 5 1
5 4 1
5 6 1
6 7 1
2 7 1
```

Výstup

```
NEEXISTUJE
```

A-III-5 Robot

Zuzka si nedávno postavila malého robota. Ovládať ho vie jedine tak, mu zadá postupnosť príkazov. Tú robot potom dokola opakuje, až kým ho Zuzka nevypne. Robot pozná len štyri príkazy: otočenie o 90° doľava, doprava, krok vpred a krok vzad.

Zuzka si vymyslela program, ktorý zadala robotovi. Potom na stole nakreslila štvorcovú sieť tak, aby veľkosť políčka zodpovedala dĺžke kroku robota. Robot je maličký, celý sa zmestí na jedno políčko. Na niektoré políčka umiestnila Zuzka prekážky. Ak sa robot pokúsi vykonať krok na políčko s prekážkou, tá ho zastaví, takže zostane na políčku, na ktorom bol. (A samozrejme pokračuje vykonaním ďalšieho príkazu v postupnosti.)

Teraz by ju zaujímalo, kam môže postaviť robota tak, aby zo stola nikdy nespadol.

Súťažná úloha:

Predpokladajme, že stôl otočíme tak, aby boli jeho strany rovnobežné so svetovými stranami. Zuzka ukladá robota na stôl vždy tak, aby sa pozeral na sever. Robota môže samozrejme položiť len na prázdne políčko. Ak robota položí na políčko, zapne ho, a ten časom zo stola spadne, hovoríme, že jeho štartovacie políčko bolo *nebezpečné*.

Napíšte program, ktorý načíta postupnosť Zuzkinych príkazov a popis stola a spočíta, koľko je na stole nebezpečných políčok.

Navyše by váš program mal pre každé nebezpečné políčko spočítať, koľko príkazov vykoná robot, ktorý na ňom začína, kým spadne zo stola. (Do tohto počtu rátame aj príkazy „krok vpred/vzad“ pri vykonaní ktorých sa robot nepohol.)

Formát vstupu:

V prvom riadku vstupu je celé číslo L ($1 \leq L \leq 500$) udávajúce počet príkazov, ktoré Zuzka robotovi zadala.

V druhom riadku je reťazec L znakov popisujúci túto postupnosť. Znaky `L` a `R` predstavujú otočenie doľava a doprava, znaky `+` a `-` predstavujú krok o políčko vpred a vzad. (To, ktorým smerom je „vpred“, samozrejme závisí od aktuálneho natočenia robota.

V treťom riadku sú dve medzerou oddelené celé čísla W a H ($1 \leq W, H \leq 500$) udávajúce rozmery stola. V smere zo západu na východ (resp. vo vstupe vodorovne) je na stole W políčok, v smere zo severu na juh (vo vstupe zvisle) ich je H.

V posledných H riadkoch vstupu je mapa stola. Každý riadok obsahuje W znakov, a každý znak je buď . (bodka, voľné políčko), alebo # (mriežka, obsadené políčko). Prvý znak v prvom riadku predstavuje severozápadný roh mapy.

Formát výstupu:

V prvom riadku vypíšte jedno celé číslo – počet nebezpečných políčok.

Následne vypíšte H riadkov a v každom z nich W medzerami oddelených čísel. Presnejšie, j-te číslo v i-tom z týchto riadkov má byť počet príkazov, ktoré robot začínajúci na tomto políčku vykoná, kým spadne zo stola. Ak políčko nie je nebezpečné alebo obsahuje prekážku, vypíšte namiesto toho nulu.

Príklad 1:

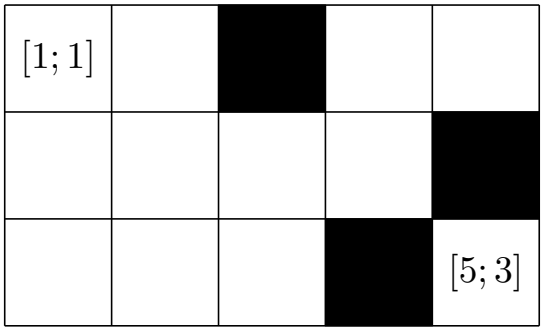
Vstup

```
16
+R++R+L+++R+R++R
5 3
..#..
....#
...#.
```

Výstup

```
7
1 1 0 1 1
0 0 0 4 0
12 0 0 0 3
```

Grafické znázornenie plánu stola (ktorý je v oboch príkladoch rovnaký):



Príklad 2:

Vstup

```
8
+R++LL+R
5 3
..#..
....#
...#.
```

Výstup

```
9
1 1 0 1 1
9 9 0 4 0
17 0 0 0 3
```

Nezabudnite, že robot danú postupnosť príkazov dokola opakuje.

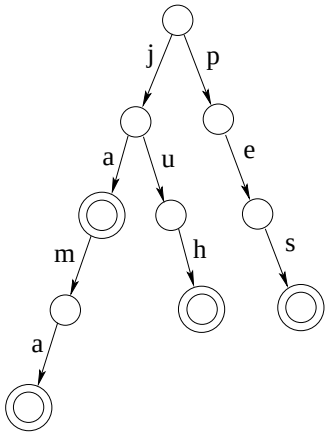
Riešenia domáceho kola kategórie A

A-I-1 Meníme históriu

Myšlienka riešenia bude jednoduchá – postupne budeme prechádzať riadky na vstupe a nahradzovať nevhodné slová vhodnými. Pri nahradzovaní slov si však musíme dať pozor, aby sme to robili efektívne.

Jedno možné efektívne riešenie je použiť dátovú štruktúru nazývanú písmenkový strom (po anglicky *trie*). Písmenkový strom je zakorenený strom, v ktorom každý vrchol má najviac 26 synov, a hrany do synov sú označené rôznymi písmenkami (od `a` po `z`). Každá cesta z koreňa nadol zodpovedá slovu, ktoré si „prečítame“ na hranách, po ktorých ideme.

Písmenkový strom sa dá použiť na uloženie množiny slov. Jednoducho vytvoríme všetky vrcholy, ktoré treba na to, aby sme si mohli na ceste z koreňa dole „prečítať“ každé z našich slov, a následne označíme tie vrcholy, kde niektoré z uložených slov končí. (Napríklad si do toho vrcholu napíšeme poradové číslo slova, ktoré sa tam končí.)



Písmenkový strom pre `ja`, `jama`, `juh` *a* `pes`.

Našu úlohu teda môžeme vyriešiť tak, že vytvoríme písmenkový strom pre zadané nevhodné slová. Pre každé spracovávané slovo zistíme, či sa v strome nachádza (teda či je nevhodné). Ak nie, nechávame ho bezo zmeny. Ak áno, nahradíme ho ekvivalentom.

Všimnime si, že pre slovo dĺžky d bude trvať vyhľadávanie v strome čas $O(d)$. Navyše výstup môže byť oveľa dlhší ako vstup. Totiž krátke nevhodné slovo môže byť nahradené dlhým vhodným slovom. Časová zložitosť je preto

lineárna od veľkosti vstupu a výstupu. Čo sa pamäťovej zložitosti týka, stačí si pamätať len posledné načítané slovo a celý písmenkový strom aj s vhodnými náhradami. Teda pamäťová zložitosť je lineárna od veľkosti stromu, čiže od veľkosti slovníkovej časti vstupu.

Listing programu:

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

class trie {
    class __trieNode { // __trieNode obsahuje vrchol pismenkoveho stromu
        public:
            int endsHere; // cislo slova, ktore sa tu konci
            __trieNode *son[26]; // pointre na synov
            __trieNode() { endsHere=-1; memset(son,0,sizeof(son)); }
    };

    __trieNode *root; // v ramci pismenkoveho stromu si pamatame pointer na koren
    public:
    trie() { root = new __trieNode(); }
    void insert(const string &S, int cislo);
    int find(const string &S);
};

// funkcia na vlozenie noveho slova do stromu
void trie::insert(const string &S, int cislo) {
    __trieNode *kde = root;

    // ideme postupne po pismenach
    for (unsigned i=0; i<S.size(); i++) {
        int idx = S[i]-'A';
        // ak este takuto vetvu v strome nemame, vytvorime si ju
        if (! kde->son[idx]) kde->son[idx] = new __trieNode();
        // a presunieme sa o uroven hlbsie
        kde = kde->son[idx];
    }
    // v aktualnom vrchole zapiseme cislo slova
    kde->endsHere = cislo;
}

// funkcia na najdenie slova v strome
int trie::find(const string &S) {
    __trieNode *kde = root;

    // opaet ideme postupne po pismenach dole stromom
    for (unsigned i=0; i<S.size(); i++) {
```

```
        if (!kde) return -1;
        kde = kde->son[ S[i]-'A' ];
    }
    if (!kde) return -1;
    return kde->endsHere;
}

int main() {
    int N; cin >> N;
    vector<string> vhodne(N);
    trie T;
    // vytvorime pismekovy strom z nevhodnych slov
    for (int i=0; i<N; i++) {
        string nevhodne;
        cin >> nevhodne >> vhodne[i];
        T.insert(nevhodne, i);
    }
    // ideme opravovat text
    string s;
    getline(cin, s); // nacitame koniec riadku za poslednym vhodnym slovom
    while (getline(cin, s)) {
        for (unsigned i=0; i<s.size(); i++) {
            if (isalpha(s[i])) { //znak je pismeno, takze sa tu zacina slovo
                unsigned u = i+1;
                while (u < s.size() && isalpha(s[u])) u++; //najdeme koniec slova
                string slovo = s.substr(i, u-i);
                i = u - 1;
                int index = T.find(slovo);
                if (index == -1) cout << slovo; else cout << vhodne[index];
            }
            else cout << s[i]; // znak nie je pismeno, rovno ho vypiseme
        }
        cout << endl;
    }
}
```

A-I-2 Prechádzka po kanáloch

Vždy, keď sa v zadaní úlohy objaví otázka „aká je najkratšia cesta“ alebo „aký je najmenší počet krokov/operácií“, prirodzeným smerom úvah je zostrojiť si graf zodpovedajúci zadanej úlohe a následne v ňom nájsť najkratšiu cestu.

V úlohách, kde hľadáme cestu v bludisku, vrcholy grafu zodpovedajú rôznym situáciám, v ktorých sa osoba idúca bludiskom (v našom prípade Jacques) môže nachádzať.

Čo v našom prípade jednoznačne popisuje Jacquovu situáciu? Samozrejme, sú to súradnice políčka, na ktorom sa nachádza. To ale nestačí – musíme ešte uviesť, ktoré kľúče už má pri sebe a ktoré nie.

Zistili sme teda, že v každom okamihu sa Jacques nachádza v jednej z $16RS$ možných situácií. (Je na jednom z RS políčok, a má so sebou jednu z $2^4 = 16$ možných kombinácií kľúčov.) Vždy sa môže skúsiť pohnúť jedným zo štyroch smerov, a zakaždým je jednoznačne určené, do akej novej situácie sa dostane.

Skutočne teda môžeme reprezentovať náš problém ako graf, kde vrcholy sú jednotlivé situácie a orientované hrany predstavujú kroky Jacquesa. V tomto grafe chceme nájsť (jednu ľubovoľnú) najkratšiu cestu zo začiatočnej situácie do ľubovoľnej situácie, kde Jacques stojí na políčku s pokladom.

Prehľadávanie do šírky:

Štandardnou metódou na hľadanie najkratšej cesty v grafe s neohodnotenými vrcholmi je prehľadávanie do šírky. Princíp je veľmi jednoduchý. Budeme postupne „ofarbovať“ všetky situácie, do ktorých sa vieme dostať. Pre každú z nich si budeme pamätať, na koľko krokov sa do nej dá dostať, a z ktorej situácie sme sa tam prvýkrát dostali.

Budeme mať frontu, v ktorej čakajú situácie na spracovanie. Na začiatku zoberieme začiatočnú situáciu, nastavíme jej vzdialenosť na 0 a vložíme ju do fronty. Kým sa nám fronta nevyprázdni, opakujeme: Vyberieme z fronty situáciu. Postupne vyskúšame všetky (nanajvýš 4) možné ťahy, ktoré sa v danej situácii dajú spraviť. Dostaneme nové situácie. Tie, ktoré sme už predtým objavili (sú „ofarbené“), odignorujeme. Ostatné „ofarbíme“, nastavíme im potrebný počet krokov o jedno väčší ako má práve spracúvaná situácia, a vložíme ich do fronty na spracovanie.

Všimnime si, ako tento algoritmus prebieha. Najskôr spracujeme začiatočnú situáciu, potom postupne všetky, ktoré sa dajú dosiahnuť na jeden krok, potom všetky dosiahnuteľné na dva kroky, a tak ďalej. A práve vďaka tomu si môžeme byť istí, že akonáhle sa prvýkrát dostaneme do nejakej novej situácie, dostali sme sa do nej určite najmenším možným počtom krokov.

Keďže každú situáciu najviac jedenkrát vložíme do fronty, a spracovať situáciu vieme v konštantnom čase (keďže sú vždy len 4 možné ťahy), je časová zložitosť tohto algoritmu lineárna od počtu dosiahnuteľných situácií.

Implementácia:

Najťažšou časťou tejto úlohy bola bezbolestná implementácia vyššie popísaného postupu.

V prvom rade je vhodné zvoliť si číslovanie políčok od 0. Potom môžeme jednoducho vypočítať nový riadok a stĺpec pri pohybe pomocou operácie modulo.

Ďalším problémom je, že vopred nevieme rozmery mapy, a mapa $1\,000\,000 \times 1\,000\,000$ sa nám do pamäte nezmestí. Priamočiare riešenie bolo použiť jednorozmerné pole, a celý popis situácie vždy zakódovať do jedného čísla. Ale výhodnejšie a omnoho prehľadnejšie je použiť dynamické polia – `vector` v C++, prípadne použiť `SetLength` vo FreePascale:

```
type pole = array of array of longint;
var R,S : longint;
    A : pole;
begin
   readln(R,S);
   setlength(A,R,S);
end.
```

Pri skúšaní všetkých možných smerov pohybu je dobré vyhnúť sa „copy and paste“ – načo mať štyri skoro rovnaké kusy programu, ak to zvládne jeden? A navyše hlavnou nevýhodou prístupu „copy and paste“ je, že veľmi pravdepodobne zabudnete na jednom mieste spraviť potrebné zmenu. A takéto chyby sa veľmi ťažko hľadajú.

V našom programe deklarujeme pomocné polia, ktoré popisujú smery, kam sa môžeme pohnúť:

```
// pomocne polia
int dr[] = {-1,1,0,0}, ds[] = {0,0,-1,1}; string ddir = "SJZV";

// sme na policku (r,s), vyskusame vsetky mozne smery:
for (int d=0; d<4; d++) {
  // riadok sa zmeni o dr[d], stlpec o ds[d], tento pohyb sa vola ddir[d]
  int nr=(r+dr[d]+R)%R, ns=(s+ds[d]+S)%S;
  // ...
}
```

Ďalej, v našej implementácii používame pri práci s dverami a kľúčmi bitové operácie pre zjednodušenie zápisu. Jednotlivé farby zodpovedajú v našom programe hodnotám 1, 2, 4 a 8. Teraz každé číslo od 0 do 15 zodpovedá jednej množine kľúčov. (Hodnota 0 znamená, že nemáme žiadne kľúče, hodnota $13 = 1 + 4 + 8$ znamená, že máme kľúče prvej, tretej a štvrtej farby.)

Teraz napríklad keď stretneme dvere, vieme ľahko zistiť, či máme zodpovedajúci kľúč. (Zoberieme bitový doplnok čísla predstavujúceho kľúče, ktoré máme, a spravíme jeho bitový and s číslom dverí. Ak dostaneme výsledok 0, dvere vieme otvoriť, ak nie, nie.)

Listing programu:

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

typedef vector<int> pole1d;
typedef vector<pole1d> pole2d;
typedef vector<pole2d> pole3d;

string sdvere="CZMF", skluce="czmf";
int dr[] = {-1,1,0,0}, ds[] = {0,0,-1,1}; string ddir = "SJZV";

int main() {
  int R, S; cin >> R >> S;
  pole2d stena(R,S), poklad(R,S), dvere(R,S), kluce(R,S);
  int startR, startS, cielR, cielS, cielK;
  bool found=false;

  for (int r=0; r<R; r++) {
    string riadok; cin >> riadok;
    for (int s=0; s<S; s++) {
      if (riadok[s]=='#') stena[r][s]=1;
      if (riadok[s]=='$') poklad[r][s]=1;
      if (riadok[s]=='*') startR=r, startS=s;
      if ((int)sdvere.find(riadok[s])>=0)
        dvere[r][s] = 1 << sdvere.find(riadok[s]);
      if ((int)skluce.find(riadok[s])>=0)
        kluce[r][s] = 1 << skluce.find(riadok[s]);
    }
  }
  pole3d dist(R,pole2d(S,pole1d(16,987654321)));
  pole3d from(R,pole2d(S,pole1d(16,-1)));
  dist[startR][startS][0] = 0;
  queue<int> Q;
  Q.push(startR); Q.push(startS); Q.push(0);
  while (!Q.empty()) {
    int r = Q.front(); Q.pop();
    int s = Q.front(); Q.pop();
    int k = Q.front(); Q.pop();
    if (poklad[r][s]) { cielR=r; cielS=s; cielK=k; found=true; break; }
    for (int d=0; d<4; d++) {
      int nr=(r+dr[d]+R)%R, ns=(s+ds[d]+S)%S;
      int nk=k | kluce[nr][ns];
      if (stena[nr][ns]) continue;
      if (dvere[nr][ns] & ~nk) continue;
      if (dist[nr][ns][nk] <= dist[r][s][k]+1) continue;
      dist[nr][ns][nk]=dist[r][s][k]+1;
```

```
      from[nr][ns][nk]=16*R*S*d + R*S*k + S*r + s;
      Q.push(nr); Q.push(ns); Q.push(nk);
    }
  }
  if (!found) { cout << "nemozne" << endl; return 0; }
  string res = "";
  while (from[cielR][cielS][cielK] >= 0) {
    int f = from[cielR][cielS][cielK];
    int s=f%S; f/=S;
    int r=f%R; f/=R;
    int k=f%16; f/=16;
    res += ddir[f]; cielR=r; cielS=s; cielK=k;
  }
  reverse(res.begin(),res.end());
  cout << res << endl;
}
```

A-1-3 Horská dráha

Pomalé riešenia:

Všetkých možných súvislých podpostupností je $O(N^2)$, pretože každá postupnosť začína na niektorom z N prvkov a končí na niektorom z nasledujúcich. Ak všetky možnosti priamočiaro vyskúšame, dostaneme čas $O(N^3)$, pretože zistenie súčtu pre nejakú podpostupnosť trvá $O(N)$ operácií.

Užitočná a často používaná finta, ktorou si môžeme pomôcť aj v tomto príklade, spočíva vo vytvorení poľa S (nazývaného *prefixové súčty*), v ktorom si budeme pre každý prvok pamätať súčet od začiatku postupnosti po tento prvok. (Navyše sa dohodneme, že $S[0] = 0$.)

Toto pole dokážeme získať postupným počítaním v lineárnom čase. A načo nám bude? V prípade, že potom budeme chcieť vedieť súčet prvkov d_i, d_{i+1}, …, d_j, nebudeme potrebovať lineárny čas, ale len použijeme hodnotu $S[j] - S[i-1]$, teda súčet $d_1, \ldots, d_j$ mínus súčet $d_1, \ldots, d_{i-1}$. Takto dostaneme riešenie pôvodnej úlohy v kvadratickom čase.

Pomocná úloha:

Pred ďalším vysvetľovaním spravíme krátku odbočku a zamyslíme sa nad nasledovnou úlohou: pre neklesajúcu postupnosť čísel $a_1, \ldots, a_N$ chceme zistiť, či v nej existujú dve čísla, ktorých rozdiel je $X, X > 0$. Úloha sa dá riešiť v lineárnom čase *metódou dvoch ukazovateľov*. Využijeme dve premenné, i a j, ktoré budú ukazovať na niektorý prvok postupnosti A, teda budú nadobúdať hodnoty indexov tejto postupnosti. Na začiatku bude ukazovateľ i nastavený na

1 a ukazovateľ j nastavený na 2. Ukazateľov budeme v texte niekedy nazývať aj *bežcami*, pretože sa na ne da pozrieť ako na dvoch ľudí, ktorí behajú po postupnosti a hľadajú riešenie.

Algoritmus bude postupovať tak, že si najprv pozrie hodnotu $a_j - a_i$. Môžu nastať tri prípady:

- $a_j - a_i$ je rovné X. Vtedy môžeme úspešne skončiť s odpoveďou *ÁNO*.
- $a_j - a_i$ je menšie ako X. Vtedy vieme, že ak existujú čísla s rozdielom X, potom musí to väčšie byť ďalej ako na indexe j. Preto budeme postupovať tak, že zvýšime ukazovateľ j o jedna. V prípade, že by j malo presiahnuť N, skončíme s odpoveďou *NIE*.
- $a_j - a_i$ je väčšie ako X. Vtedy vieme, že ak existujú dve čísla s rozdielom X, potom musí to menšie z nich byť ďalej ako na indexe i. Preto budeme postupovať tak, že zvýšime ukazovateľ i o jedna.

Prečo uvedeným správaním dostaneme korektné riešenie? Odpovedať môžeme pomocou matematickej indukcie ukázaním, že po k krokoch sme nemohli minúť riešenie, ktorého väčší prvok je v poli skorej ako aktuálna pozícia bežca j a zároveň sme nemohli minúť riešenie, ktorého pozícia menšieho prvku je skorej ako pozícia bežca i. Po nula krokoch to zjavne platí, pretože keďže $X > 0$, prvky, ktorých rozdiel hľadáme, musia byť rôzne a preto má ten väčší pozíciu aspoň 2. Uvažujme, že po k krokoch nastal prípad druhý prípad, teda $a_j - a_i < X$. Vieme, že žiadne riešenie nemá vyšší prvok skôr ako je aktuálna pozícia bežca j, obdobne neexistuje riešenie, ktorého nižší prvok je skôr ako pozícia čísla i. Takže ak existuje riešenie, najmenšie pozície môže mať i a j. Ale keďže $a_j - a_i < X$, potom indexy i a j nie sú riešením a vzhľadom k predpokladu musí prípadné riešenie mat vyšší prvok na pozícii minimálne $j + 1$. Preto môžeme príslušný ukazovateľ posunúť a uvedená vlastnosť bude platiť aj po $k + 1$ krokoch. Obdobne sa dokáže aj tretia možnosť. V prípade, že j presiahne rozsah poľa, naozaj môžeme skončiť s neúspechom, pretože prípadné riešenie by muselo končiť mimo postupnosti a teda nemôže existovať.

Časová zložitosť tohto postupu je $O(N)$, pretože každý krok trvá konštantný čas a v každom kroku posunieme jedného bežca, takže ich robíme najviac $2N$. Zamyslite sa, ako nám pri tomto riešení pomáha neklesajúcosť postupnosti $a_1, \ldots, a_N$.

Ľahkou modifikáciou uvedeného postupu sa dajú riešit aj iné úlohy. Jednou z nich je zistiť, či je v postupnosti nezáporných čísel súvislá podpostupnosť

s nejakým zadaným súčtom. Toto nám pripomína úlohu zo zadania. A naozaj, keby sme sa obmedzili na nezáporne čísla, vedeli by sme napísať lineárne riešenie. So zápornými číslami sa ale tento postup nedokáže vysporiadať.

Riešenie v čase $O(N \log N)$:

Vieme už, že keď si predpočítame pole S, vieme súčet úseku $d_i, \ldots, d_j$ vyjadriť ako $S[j] - S[i-1]$. Našu úlohu teda môžeme preformulovať nasledovne: Nájdite indexy a a b také, že $|S[a] - S[b]|$ je čo najbližšie k X. Keďže a a b môžeme medzi sebou vymeniť, stačí nájsť dvojicu a, b takú, že $S[a] - S[b]$ je čo najbližšie k X. A toto nám už začína pripomínať našu pomocnú úlohu.

Ak by sme mali len zistiť, či sa dá dosiahnuť $S[a] - S[b] = X$, už by sme vedeli ako na to. Stačilo by utriediť pole S podľa veľkosti a následne použiť metódu dvoch ukazovateľov. A teraz si už len stačí uvedomiť, že táto metóda dokonca bez akejkoľvek zmeny vyrieši aj našu všeobecnejšiu úlohu.

Prečo je to tak? Nech Y je najväčšia hodnota menšia ako X, ktorá sa dá dosiahnuť. Keby sme teraz pustili ten istý algoritmus, ale nechali ho hľadať Y namiesto X, jeho priebeh by bol celkom rovnaký – všetky porovnávania by dopadli rovnako, a teda by rovnako posúval ukazovatele. No a vyššie sme dokázali, že ak sa hodnota Y dá dosiahnuť, náš algoritmus to zistí. Inými slovami, keď budeme hľadať rozdiel X, určite bude medzi skúmanými dvojicami indexov aj tá dvojica, pre ktorú je rozdiel Y.

Podobne sa dá dokázať, že počas použitia metódy dvoch ukazovateľov nájdeme aj najmenšiu hodnotu Z, ktorá je väčšia ako X a dá sa dosiahnuť. A na vyriešenie našej úlohy stačí jednoducho vybrať tú lepšiu z týchto dvoch možností.

Ešte raz teda zhrnieme celé riešenie: Predpočítame si prefixové súčty do poľa S. Toto pole následne utriedime. Teraz v ňom chceme nájsť dve hodnoty $S[a]$ a $S[b]$ také, že $S[a] - S[b]$ je čo najbližšie k X. To spravíme tak, že na utriedené pole S spustíme algoritmus dvoch ukazovateľov – postupne skracujeme alebo predlžujeme skúmaný interval, podľa toho, či je aktuálna hodnota rozdielu menšia alebo väčšia ako hľadané X. Nakoniec spomedzi všetkých dvojíc indexov, ktoré náš algoritmus skúmal, vyberieme najlepšiu a vypíšeme ju.

Poznámka k implementácii: Aby sme dokázali nájsť indexy k, l, ktoré nás vo výstupe zaujímajú, tak si ku každému prvku $S[i]$ zapamätáme aj index i, ktorý potom presúvame pri triedení spolu s hodnotou $S[i]$.

Predrátanie poľa S nás stojí čas $O(N)$. Triedenie pomocou efektívneho algoritmu trvá $O(N \log N)$ a algoritmus dvoch ukazovateľov potrebuje čas $O(N)$. Výsledná časová zložitosť je $O(N \log N)$, pamäťové nároky sú zjavne $O(N)$.

Iné riešenie v čase $O(N \log N)$:

Namiesto metódy dvoch ukazovateľov stačilo použiť efektívnu dátovú štruktúru, do ktorej vieme vkladať prvky, a pre danú hodnotu q nájsť prvok, ktorého hodnota je najbližšia ku q. Pomocou takejto dátovej štruktúry vieme zadanú úlohu ľahko vyriešiť. Pripomeňme, že hľadáme indexy a a b také, že $S[a] - S[b]$ má byť čo najbližšie k X. Začneme tým, že do našej dátovej štruktúry vložíme všetky hodnoty $S[i]$. Teraz postupne vyskúšame všetky možné hodnoty a, a pre každú si v našej dátovej štruktúre nájdeme prvok s hodnotou najbližšou ku $S[a] - X$.

Programátori v C++ majú k dispozícii vyvážený binárny strom – triedu `set`. V Pascale jednoduchou alternatívou bolo utriediť pole S a binárne v ňom vyhľadávať. Oba prístupy dokážu hľadaný prvok nájsť v čase $O(\log N)$, a teda celková časová zložitosť takýchto riešení je $O(N \log N)$.

Listing programu:

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

int main(){
    int N,X;
    cin >> N >> X;
    vector<int> A(N);
    for (int i=0;i<N;i++) cin >> A[i];

    // spocitame ciastkove sucty od zaciatku po kazdy prvok postupnosti
    // prva zlozka prvku S[i] je sucet od zaciatku postupnosti po prvok A[i-1]
    // druha zlozka prvku S[i] je samotny index i, ktoru potrebujeme pri vystupe
    vector<pair<int,int> > S(N+1);
    S.resize(N+1);
    S[0] = make_pair(0,0);
    for (int i=1;i<=N;i++) S[i] = make_pair( S[i-1].first+A[i-1], i );
    sort(S.begin(),S.end());

    //nastavime ukazovatele i,j na zaciatok
    int i=0, j=1, suc, besti=S[0].second, bestj=S[1].second,
        bestsuc=S[1].first-S[0].first;

    while ( i<= N && j<= N ){
        suc = S[j].first - S[i].first;
        //nasli sme lepsiu moznost ako doteraz najlepsiu?
        if (abs(X-suc) < abs(X-bestsuc)) {
          besti=S[i].second; bestj=S[j].second; bestsuc=suc;
        }
```

```
        //ak sme nasli najlepsiu moznu odpoved, mozeme skoncit
        if (X == suc) break;
        if (X > suc) j++; else i++;
        //prazdnu postupnost nepripustame
        if (i == j) j++;
    }
    if (besti > bestj) swap(besti,bestj);
    cout << besti+1 << " " << bestj << endl;
    return 0;
}
```

A-1-4 Zásobníkové počítače

Podúloha a:

V tejto podúlohe sme mali dosiahnuť čo najlepšiu časovú zložitosť. Naše riešenie je lineárne a používa tri zásobníky. Menší počet operácií ako lineárny od dĺžky vstupu dosiahnuť zjavne nemôžeme, pretože na správnu odpoveď musíme načítať celé slovo.

Ak by sme mali dva zásobníky, z ktorých v jednom by bolo uložené vstupné slovo v poradí, ako sme ho načítali a v druhom by bolo uložené v opačnom poradí (teda na vrchu by bol znak, ktorý sme načítali ako prvý), potom by sa nám skutočnosť, či je vstup palindróm, overovala veľmi jednoducho – len by sme čítali znaky zo zásobíkov a pozerali sa, či sú rovnaké.

Riešenie začne tým, že načíta vstup, ktorý si zapamätá do dvoch zásobníkov – $S1$ a $S2$. Potom využije ďalší prázdny zásobník $S3$, do ktorého „presype“ jeden z naplnených – bude čítať znaky z prvého zásobníka a ukladať ich na druhý. Takto získame v zásobníku $S3$ slovo zo vstupu v opačnom poradí. A teraz už stačí len postupne overiť rovnosť všetkých dvojíc písmen.

Načítanie vstupu, preklopenie zásobníka aj overovanie trvá lineárny počet operácií od dĺžky vstupného slova.

Listing programu:

```
program palindrom1;
var S1,S2,S3: stack of char;
    c,d: char;
begin
   while read(c) do begin push(S1, c); push(S2, c); end;
   {naplnenie S3, aby v nom bol vstup opacne ako v S2}
   while not empty(S1) do begin c := pop(S1); push(S3,c); end;
   while not empty(S2) do begin
      c = pop(S2);
```

```
      d = pop(S3);
      if (c <> d) then begin write('0'); halt; end;
   end;
   write('1');
end.
```

Podúloha b:

Naše riešenie bude používať dva zásobníky - $S1$ a $S2$. Pomocou nich najprv overí, či sa prvý znak zhoduje s posledným, potom či sa druhý zhoduje s predposledným a tak postupne až kým neskontroluje všetky protiľahlé dvojice písmen v slove.

Program na začiatku načíta celý vstup do zásobníka $S1$. Následne vyberie prvý znak a zapamätá si ho. Potom celý zvyšný obsah $S1$ presype do $S2$ - postupne číta znaky z vrchu $S1$ a ukladá ich na vrch zásobníka $S2$. Keď to spraví, je pripravený overiť si zhodu prvého a posledného znaku. Ak sa nerovnajú, môže hneď prehlásiť, že slovo nie je palindróm. V prípade rovnosti znakov „presypeme“ celý zvyšok slova späť do $S1$ a celý postup začneme odznova.

Ak je dĺžka vstupného slova N, riešenie vykoná pre každú overovanú dvojicu $O(N)$ operácií. Overovaných dvojíc je približne $N/2$, preto je výsledná časová zložitosť $O(N^2)$.

Listing programu:

```
var S1,S2: stack of char;
    c,d: char;
begin
   { nacitame vstup }
   while read(c) do push(S1,c);
   while true do begin
      { ak uz nezostali ziadne pismena, je to palindrom }
      if empty(S1) then begin write(1); halt; end;
      { zoberieme pismeno z vrchu S1, zvysok S1 presypeme do S2 }
      c := pop(S1);
      while not empty(S1) do begin d:=pop(S1); push(S2,d); end;
      { ak je S2 prazdny, ostalo len jedno pismeno, je to palindrom }
      if empty(S2) then begin write(1); halt; end;
      { zoberieme pismeno z vrchu S2, porovname so zapamatanym z opacneho konca }
      d := pop(S2);
      if c<>d then begin write(0); halt; end;
      { "presypeme" cely obsah S2 spat a cely postup opakujeme }
      while not empty(S2) do begin d:=pop(S2); push(S1,d); end;
   end;
end.
```

Riešenia domáceho kola kategórie B

B-I-1 Diaľnica

Pomalšie riešenia:

Najjednoduchšie, priamočiare riešenie spočíva v prehľadaní všetkých dvojíc čerpacích staníc. Pre každú dvojicu staníc si vypočítame ich vzdialenosť, a ak je menšia alebo rovná K, zväčšíme si počítadlo.

```
P := 0;
for i := 1 to N do
   for j := i + 1 to N do
      if(a[j] - a[i] <= K) then P := P + 1;
writeln(P);
```

Všetkých dvojíc staníc je $N(N-1)/2$, čo je rádovo N^2. Preto má takéto riešenie časovú zložitosť $O(N^2)$. Malo získať aspoň 5 bodov.

Ak sa podrobnejšie pozrieme na predchádzajúce riešenie, všimneme si, že konštrukcia dvoch cyklov `for` nám generuje všetky dvojice čerpacích staníc – teda aj tie, ktorých vzdialenosť je väčšia ako K. Zjavne ale platí, že akonáhle už pre nejaké j je stanica j príliš daleko od stanice i, tak aj všetky stanice od $j+1$ po N sú už pridaleko. Vnútorný cyklus teda môžeme prerušiť akonáhle vzdialenosť od stanice i presiahne hodnotu K.

```
P := 0;
for i := 1 to N do
   for j := i + 1 to N do
      if(a[j] - a[i] <= K) then P := P + 1
      else break;
writeln(P);
```

Tým dostaneme riešenie, ktorého časová zložitosť je $O(N+P)$, kde P je hodnota výsledku. Takéto riešenie je lepšie od predchádzajúceho pre „riedke“ vstupy, avšak v najhoršom prípade (ak sú každé dve stanice vo vzdialenosti $\leq K$) je jeho časová zložitosť nadalej kvadratická od N. Toto riešenie malo získať aspoň 7 bodov.

Vzorové riešenie:

Ak chceme riešenie, ktoré bude mať vždy lepšiu ako kvadratickú časovú zložitosť, nesmieme sa „dotknúť“ všetkých hľadaných dvojíc, kedže tých môže byť až kvadraticky veľa.

Čo ešte vieme na predchádzajúcom riešení zlepšiť? Nech m_i je najväčšie číslo stanice, ktorá je dosiahnuteľná zo stanice i. Keď teraz budeme spracúvať stanicu

$i+1$, určite budú aj z nej dosiahnute stanice $i+2$ až m_i, lebo sú k nej bližšie ako ku stanici i. Tieto dvojice staníc teda nemusíme kontrolovať a môžeme ich rovno započítať do výsledku.

Iný možný pohľad na toto isté pozorovanie: Práve sme zdôvodnili, že platí $m_{i+1} \geq m_i$. Zjavne platí, že hodnota $m_i - i$ udáva počet staníc, ktoré sú napravo od i a sú z nej dosiahnuteľné. Preto výsledkom našej úlohy je $P = \sum_{i=1}^{n} m_i - i$. Na to, aby sme zistili P, teda stačí spočítať hodnoty m_i.

Listing programu:

```
const maxN = 210000;
var N,K,P,i : longint;
    a,m : array[0..maxN] of longint;
begin
   read(K); read(N); for i:=1 to N do read(a[i]);
   P := 0;
   m[0] := 1;
   for i:=1 to N do begin
      m[i] := m[i-1];
      while (m[i] < N) and (a[ m[i]+1 ] - a[i] <= K) do inc(m[i]);
      P := P + m[i] - i;
   end;
   writeln(P);
end.
```

Všimnime si, že aktuálna hodnota m_i sa v priebehu programu nikdy nezmenší. Preto sa príkaz `inc(m[i])` počas celého programu vykoná najviac $N-1$ ráz. A preto je aj celková časová zložitosť tohto riešenia lineárna od N.

(Na riešenie sa môžeme dívať tak, že si na pole ukazujeme dvoma prstami – ľavým na práve spracúvanú stanicu, pravým na stanicu, s ktorou ju práve porovnávame. Počas behu programu prejdeme oboma prstami po poli zľava doprava, dokopy teda pohneme prstom menej ako $2N$-krát.)

B-I-2 Kolotoč

Najjednoduchšie riešenie úlohy spočívalo jednoducho v implementácii postupu zo zadania: Napíšeme si procedúru, ktorá celý kolotoč otočí o jedno miesto, a následne ju K-krát zavoláme. Takéto riešenie má časovú zložitosť $O(KN^2)$, keďže K-krát posunieme každé z N^2 detí. Správny program s touto myšlienkou mal získať aspoň 5 bodov.

Prvé možné zlepšenie je všimnúť si, ako vlastne ten kolotoč otáčame. Kolotoč sa skladá z približne $N/2$ „obručí“, ktoré sa točia nezávisle na sebe. Všimnime

si napríklad vonkajšiu obruč kolotoča $N \times N$. Tú tvorí $4N - 4$ sedadiel. To ale znamená, že každých $4N - 4$ sekúnd budú deti, ktoré sú na tejto obruči, sedieť presne na tých miestach ako na začiatku.

Ak je počet otočení K výrazne väčší ako N, vieme vďaka tomuto pozorovaniu dosť práce ušetriť: pre každú obruč si spočítame jej dĺžku D, a následne ju otočíme (K mod D)-krát.

Pri tomto riešení posunieme každé dieťa najviac $(4N-5)$-krát, celková časová zložitosť je teda $O(N^3)$. Program s touto myšlienkou mal získať aspoň 7 bodov.

Vzorové riešenie ide ešte o krok ďalej: keď máme obruč a vieme, koľkokrát ju chceme otočiť, môžeme rovno pre každé dieťa spočítať, kde bude sedieť po danom počte otočení.

Presnejšie, očíslujme si pozície na danej obruči číslami od 0 do $D - 1$ v smere točenia. Dieťa, ktoré teraz sedí na pozícii x, bude po K otočeniach sedieť na pozícii $(x + K)$ mod D. Stačí si teda spraviť nové pole a do neho postupne vypĺňať deti na miesta, kde budú sedieť na konci.

Pri tomto riešení každé dieťa rovno umiestnime na pozíciu, kde bude na konci. Celková časová zložitosť je teda $O(N^2)$. Program s touto myšlienkou mal získať 10 bodov.

V nasledujúcom listingu programu si všimnite elegantný spôsob implementácie funkcie `otoc_obruc`: do polí `row` a `col` si vyplníme súradnice políčok obruče, a následne pomocou tejto informácie hravo vyplníme túto obruč vo výstupnom poli.

Listing programu:

```
const maxN = 1024;
var vstup, vystup : array[0..maxN,0..maxN] of string[8];
    N,K,i : longint;

procedure nacitaj;
var riadok : string;
    zac,kon,row,col,i : longint;
begin
   readln(N);
   for row:=0 to N-1 do begin
      { nacita a spracuje riadok mien }
      readln(riadok); riadok:=riadok+' ';
      zac:=1; kon:=1;
      for col:=0 to n-1 do begin
         while (riadok[kon]<>' ') do inc(kon);
         vstup[row][col]:='';
         for i:=zac to kon-1 do vstup[row][col]:=vstup[row][col]+riadok[i];
         zac:=kon+1; kon:=kon+1;
```

```
      end;
   end;
   readln(K);
end;

procedure otoc_obruc(roh : longint);
var s,D,i,j : longint;
    row,col : array[0..4*maxN] of longint;

begin
   s:=n-2*roh-1; D:=4*s;
   if D=0 then begin D:=1; row[0]:=roh; col[0]:=roh; end;
   for i:=0 to s-1 do begin row[0*s+i]:=roh;     col[0*s+i]:=roh+i;   end;
   for i:=0 to s-1 do begin row[1*s+i]:=roh+i;   col[1*s+i]:=roh+s;   end;
   for i:=0 to s-1 do begin row[2*s+i]:=roh+s;   col[2*s+i]:=roh+s-i; end;
   for i:=0 to s-1 do begin row[3*s+i]:=roh+s-i; col[3*s+i]:=roh;     end;
   for i:=0 to D-1 do begin
      j:=(i+K) mod D;
      vystup[ row[j] ][ col[j] ] := vstup[ row[i] ][ col[i] ];
   end;
end;

procedure vypis;
var i,j : longint;
begin
   for i:=0 to N-1 do begin
      for j:=0 to N-1 do write(vystup[i][j]:7); writeln;
   end;
end;

begin
   nacitaj;
   for i:=0 to (N-1) div 2 do otoc_obruc(i);
   vypis;
end.
```

Uvedieme ešte veľmi stručnú implementáciu toho istého algoritmu v jazyku C++.

Listing programu:

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

int main() {
  int N,K;
  cin >> N;
  vector< vector<string> > vstup(N,N), vystup(N,N);
```

```
  for (int i=0; i<N; i++) for (int j=0; j<N; j++) cin >> vstup[i][j];
  cin >> K;
  for (int roh=0; roh<(N+1)/2; roh++) {
    int s=N-2*roh-1, D=max(1,4*s);
    vector<int> row(D), col(D);
    if (D==1) row[0] = col[0] = roh;
    for (int i=0; i<s; i++) row[0*s+i]=roh,     col[0*s+i]=roh+i;
    for (int i=0; i<s; i++) row[1*s+i]=roh+i,   col[1*s+i]=roh+s;
    for (int i=0; i<s; i++) row[2*s+i]=roh+s,   col[2*s+i]=roh+s-i;
    for (int i=0; i<s; i++) row[3*s+i]=roh+s-i, col[3*s+i]=roh;
    for (int i=0; i<D; i++)
      vystup[ row[(i+K)%D] ][ col[(i+K)%D] ] = vstup[ row[i] ][ col[i] ];
  }
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) cout << setw(7) << vystup[i][j];
    cout << endl;
  }
}
```

B-I-3 Balíčky

Pre vyriešenie úlohy stačí zistiť, ktoré balíčky budú a ktoré nebudú nainštalované. Celkovú veľkosť už potom zistíme ľahko.

Vytvorme si teda pole `nainstalovany[]` typu boolean, pričom cieľom nášho riešenia bude nastaviť hodnotu `nainstalovany[i]` na `true`, ak balíček musíme inštalovať, a na `false` inak. Na začiatku všetky hodnoty v poli `nainstalovany` nastavíme na `false`.

Úlohu budeme riešiť pomocou rekurzie. Pre inštaláciu i-teho balíčka si vytvoríme funkciu `instaluj(i)`. Táto funkcia bude mať za úlohu zabezpečiť to, aby všetky balíčky potrebné na fungovanie balíčka i mali priradenú hodnotu v poli `nainstalovany` na `true`. V nasledujúcom kóde predpokladáme, že závislosti balíčka i sú uložené v poli `zavislost[i]`, na pozíciách 1 až `p[i]`.

```
procedure instaluj(i : longint);
var j : longint;
begin
   nainstalovany[i] := true;
   for j := 1 to p[i] do
      if( not nainstalovany[zavislost[i][j]] ) then
         instaluj(zavislost[i][j]);
end;
```

Pre vyriešenie našej úlohy teraz postupne zavoláme funkciu `instaluj` pre každý balíček, ktorý chce používateľ.

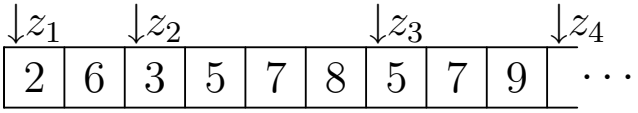
Poďme teraz analyzovať toto riešenie. Hodnota `nainstalovany[i]` sa počas behu programu nastaví na `true` práve vtedy, ak sa počas behu zavolá funkcia inštaluj s parametrom i. Všimnime si ale, že počas behu celého programu sa funkcia `instaluj` zavolá s parametrom i maximálne raz. (Akonáhle ju zavoláme, nastaví sa `nainstalovany[i]` na `true`, vďaka čomu ju už nikdy nezavoláme znova.)

Pri volaní procedúry `instaluj` s parametrom i vykonáme jedno priradenie, a následne postupne spracujeme všetkých p_i balíčkov, na ktorých balíček i závisí. Dokopy teda spravíme najviac N zmien v poli `nainstalovany`, a pri kontrolách závislostí spracujeme dokopy nanajvýš $\sum_{i=1}^{n} p_i$ balíčkov. Preto bude časová zložitosť nášho riešenia $O(N + \sum_{i=1}^{n} p_i)$. Už na načítanie vstupných údajov potrebujeme takýto čas, a preto je toto riešenie optimálne.

Poslednou vecou, ktorou sa v našom riešení potrebujeme zaoberať, je dátová štruktúra, ktorú použijeme na uchovávanie závislostí. Pre každý balíček si potrebujeme zapamätať zoznam balíčkov, na ktorých závisí. Tento zoznam ale môže obsahovať až $N - 1$ balíčkov. Ak by sme si tento zoznam chceli pamätať v poli dĺžky N, pre každé i, dostali by sme maticu veľkosti $N \times N$. Takéto riešenie by nám ale zvýšilo pamäťovú (a v závislosti od implementácie možno aj časovú) zložitosť na $O(N^2)$.

Na vyriešenie tohoto problému máme niekoľko možností.

Prvou možnosťou je dynamická alokácia pamäti – vytvoriť pre každý balíček pole dĺžky rovnej počtu jeho závislostí. Druhým možným riešením by bolo na uloženie závislostí implementovať vhodnú vlastnú dátovú štruktúru (napr. spájaný zoznam). Treťou možnosťou je zapamätať si závislosti všetkých balíčkov v jedinom poli s. Pre každý balíček i si okrem počtu jeho závislostí p_i budeme tiež pamätať index z_i, ktorý bude určovať, kde v poli s začína zoznam závislostí pre balíček i. Čísla balíčkov, na ktorých závisí balíček i sa teda budú nachádzať na pozíciách $z_i, z_i + 1, z_i + 2, \ldots, z_i + p_i - 1$. Nasledujúci obrázok vykresľuje nami predstavené riešenie, kde prvý balíček závisí na balíčkoch 2 a 6, druhý balíček závisí na balíčkoch 3,5,7,8 a tretí balíček závisí na balíčkoch 5,7,9.



Implementáciu takejto štruktúry používame aj v našom programe.
A ešte otázka na záver: Fungovalo by toto riešenie aj vtedy, keby mohli byť v zozname závislostí aj cykly?

Listing programu:

```
var N,M,i,j,q:longint;
    c:char;
    { velkosti balickov, pocty zavislosti, zaciatky v poli s }
    v,p,z:array[0..10000] of longint;
    { pole obsahujuce zavislosti vsetkych balickov }
    s:array[1..100000] of longint;
    nainstalovany:array[1..10000] of boolean;
    vysledok:longint;

procedure instaluj(i : longint);
var j : longint;
begin
   nainstalovany[i] := true;
   for j := 1 to p[i] do
      if ( not nainstalovany[ s[z[i]+j-1] ] ) then instaluj( s[z[i]+j-1] );
end;

begin
   readln(N);
   q := 1;                          { q je prva este nezaplnena pozicia v poli s }
   for i := 1 to N do begin
      repeat read(c); until c=' '; { precitame nazov balicka a odignorujeme ho }
      read(v[i],p[i]);
      z[i] := q;
      for j := 1 to p[i] do begin  { ulozime zavislosti i-teho balicka do s }
         read(s[q]);
         inc(q);
      end;
      readln;
   end;
   for i := 1 to N do nainstalovany[i] := false;
   readln(M);
   for j := 1 to M do begin
      read(i);
      instaluj(i);
   end;
   vysledok:=0;
   for i := 1 to N do if (nainstalovany[i]) then vysledok := vysledok + v[i];
   writeln(vysledok);
end.
```

B-I-4 Divné jazyky

Podúloha a):

Túto podúlohu bolo najľahšie vyriešiť vypísaním všetkých možností. Oplatí sa pritom byť systematický – môžeme odpozorovať závislosti, ktoré nám pomôžu neskôr.

Vypíšme teda všetky symetrické slová (odborne sa im hovorí *palindrómy*) s počtom 1 až 7 písmen:

- `A`, `U`
- `AA`, `UU`
- `AAA`, `AUA`, `UAU`, `UUU`
- `AAAA`, `AUUA`, `UAAU`, `UUUU`
- `AAAAA`, `AAUAA`, `AUAUA`, `AUUUA`, `UAAAU`, `UAUAU`, `UUAUU`, `UUUUU`
- `AAAAAA`, `AAUUAA`, `AUAAUA`, `AUUUUA`, `UAAAAU`, `UAUUAU`, `UUAAUU`, `UUUUUU`
- `AAAAAAA`, `AAAUAAA`, `AAUAUAA`, `AAUUUAA`, `AUAAAUA`, `AUAUAUA`, `AUUAUUA`, `AUUUUUA`, `UAAAAAU`, `UAAUAAU`, `UAUAUAU`, `UAUUUAU`, `UUAAAUU`, `UUAUAUU`, `UUUAUUU`, `UUUUUUU`

Celkový počet vypísaných slov je 44.

Podúloha b):

Všimnime si počty symetrických slov pre jednotlivé dĺžky: máme 2 slová dĺžky 1, 2 slová dĺžky 2, 4 slová dĺžky 3, atď. Dostávame takto postupnosť: 2, 2, 4, 4, 8, 8, 16, ...

Pomerne ľahko sa dá uhádnuť, ako bude táto postupnosť pokračovať: 16, 32, 32, 64, 64, ... Tvrdíme teda, že aj symetrických slov s $2k-1$ písmenami, aj symetrických slov s $2k$ písmenami je práve 2^k.

Nasledujúcim krokom riešenia je toto pozorovanie nejak dokázať. Jeden možný dôkaz: Predstavte si, že ideme napísať symetrické slovo tvorené $2k$ písmenami. Akonáhle napíšeme prvých k písmen, je vždy práve jedna správna možnosť, ako doplniť zvyšných k – musíme napísať tie isté písmená v opačnom poradí. No a pri výbere každého z prvých k písmen máme dve možnosti (`A` alebo `U`). Preto existuje 2^k spôsobov, ako napísať prvých k písmen, a teda existuje práve 2^k symetrických slov dĺžky $2k$.

Tento dôkaz môžeme povedať aj inými slovami: Všetky symetrické slová dĺžky $2k$ vieme vyrobiť tak, že zoberieme úplne všetky slová dĺžky k (ktorých je zjavne 2^k) a za každé slovo dopíšeme dotyčné slovo odzadu.

(Rozmyslite si, čo sa v tomto dôkaze zmení, ked budeme dokazovať, že aj počet symetrických slov s $2k-1$ písmenami je 2^k.)

Riešenie tejto podúlohy je teda $B = 2+2+\cdots+2^{23}+2^{23}+2^{24} = 50331644$. Pre zaujímavosť podotkneme, že pomocou známeho vzorca pre súčet geometrického radu ľahko vyjadríme, že $B = 2^{25}+2^{24}-4$.

Podúloha d):

Kmeň Strapatých má naozaj podivuhodný jazyk. Všetky slová v ich jazyku majú presne 7 hlások. Navyše platí, že ak sa domorodec z tohto kmeňa pomýli v jednej hláske ľubovoľného slova, aj tak sa dá spoznať, ktoré slovo chcel povedať. Dokážeme, že tento jazyk môže mať najviac **16 slov**.

Ku každému slovu S v jazyku kmeňa Strapatých existuje práve 7 reťazcov, ktoré sa od neho líšia v práve jednom znaku. Týchto 7 reťazcov nazveme *kamarátmi* slova S.

Všimnime si, že medzi kamarátmi slova S nemôže byť ani žiadne iné slovo, ani kamarát žiadneho iného slova.

Predstavme si teraz, že by sme zobrali list linajkového papiera a vypísali naň slová nášho jazyka, každé do nového riadku. Následne za každé slovo dopíšeme jeho 7 kamarátov.

Je zjavné, že na našom papieri nemôže byť žiaden reťazec uvedený dvakrát.

Všetkých reťazcov dĺžky 7 je $2^7 = 128$. V každom riadku je uvedených 8 reťazcov. Preto riadkov môže byť najviac $128/8 = 16$. Ale počet riadkov je práve rovný počtu slov v našom jazyku. Tým sme dokázali, že takýto jazyk môže mať nanajvýš 16 slov.

Podúloha c):

Zoraďme si všetkých $2^7 = 128$ reťazcov podľa abecedy. V tomto poradí ich budeme prechádzať. Pre každý reťazec otestujeme, či ho môžeme pridať do jazyka (t. j., či sa od každého už vybraného slova líši v aspoň troch znakoch), a ak áno, pridáme ho tam.

V tomto okamihu si treba uvedomiť, že nie je zaručené, že nám takýto postup nájde najväčší možný jazyk. Teoreticky by sa mohlo stať, že keby sme niektorý reťazec preskočili, umožnilo by nám to neskôr pridať viac iných reťazcov. Výhodou uvedeného „pažravého“ postupu však je, že sa veľmi ľahko naprogramuje, a teda určite neuškodí vyskúšať ho. Dostaneme tento jazyk:

`AAAAAAA`, `AAAAUUU`, `AAUUAAU`, `AAUUUUA`, `AUAUAUA`, `AUAUUAU`, `AUUAAUU`, `AUUAUAA`, `UAAUAUU`, `UAAUUAA`, `UAUAAUA`, `UAUAUAU`, `UUAAAAU`, `UUAAUUA`, `UUUUAAA`, `UUUUUUU`.

Zostrojili sme teda jazyk tvorený 16 slovami. Z výsledku podúlohy d) vieme,

že lepšie to nejde. Môžeme teda spokojne prehlásiť, že sme našli najväčší možný jazyk spĺňajúci podmienky zo zadania.

V programe sme pre jednoduchšiu manipuláciu namiesto hlások `A` a `U` použili hodnoty 0 a 1. Takto sa nám 7-znakové reťazce zmenia na binárne zápisy čísel $0000000 = 0$ až $1111111 = 127$.

Ako súčasť riešenia uvádzame aj druhý program, založený na metóde prehľadávania s návratom (backtrackingu), ktorý postupne nájde všetky jazyky spĺňajúce uvedenú podmienku a vyberie najväčší z nich. Tento program vznikol jednoduchou úpravou prvého riešenia – vždy, keď nájdeme slovo, ktoré môžeme pridať do jazyka, vyskúšame postupne obe možnosti, aj pridať ho, aj nepridať. U tohto programu máme istotu, že nájde najväčšie možné riešenie.

Listing programu:

```
program b_1_4_greedy;

var slova : array[1..16] of longint;
    N, i : longint;

function vzdialenost(s1, s2 : longint) : longint;
var vysledok, i : longint;
begin
   vysledok := 0;
   for i:=0 to 6 do { porovname i-ty bit cisel s1 a s2 }
      if (s1 and (1 shl i)) <> (s2 and (1 shl i)) then inc(vysledok);
   vzdialenost := vysledok;
end;

function mozeme_pouzit(slovo : longint) : boolean;
var i : longint;
begin
   mozeme_pouzit := true;
   for i := 1 to N do if vzdialenost(slovo,slova[i]) <= 2 then
      mozeme_pouzit := false;
end;

procedure vypis(slovo : longint);
var i : longint;
begin
   for i:=6 downto 0 do
      if (slovo and (1 shl i)) > 0 then write('U') else write('A');
   writeln;
end;

begin
   N := 0;
   for i := 0 to 127 do
```

```
      if mozeme_pouzit(i) then begin inc(N); slova[N] := i; end;
   for i:=1 to N do vypis(slova[i]);
end.
```

Listing programu:

```
program b_1_4_backtracking;

var teraz, najlepsie : array[1..16] of longint;
    N, najlepsieN, i : longint;

function vzdialenost(s1, s2 : longint) : longint;
var vysledok, i : longint;
begin
   vysledok := 0;
   for i:=0 to 6 do { porovname i-ty bit cisel s1 a s2 }
      if (s1 and (1 shl i)) <> (s2 and (1 shl i)) then inc(vysledok);
   vzdialenost := vysledok;
end;

function mozeme_pouzit(slovo : longint) : boolean;
var i : longint;
begin
   mozeme_pouzit := true;
   for i := 1 to N do if vzdialenost(slovo,teraz[i]) <= 2 then
      mozeme_pouzit := false;
end;

procedure skus;
var posledne, nove, i : longint;
begin
   posledne := teraz[N];
   for nove := posledne+1 to 127 do
      if mozeme_pouzit(nove) then begin
         { pridame nove slovo do jazyka }
         inc(N); teraz[N] := nove;
         { pozrieme ci sme nasli vacsi jazyk ako doteraz najvacsi }
         if N > najlepsieN then begin
            najlepsieN := N;
            for i := 1 to N do najlepsie[i] := teraz[i];
         end;
         { skusime vsetky sposoby ako pridat dalsie slova }
         skus;
         { opat toto slovo odoberieme a skusime namiesto neho pridat ine }
         dec(N);
      end;
end;

procedure vypis(slovo : longint);
var i : longint;
begin
```

```
   for i:=6 downto 0 do
      if (slovo and (1 shl i)) > 0 then write('U') else write('A');
   writeln;
end;

begin
   N := 1;
   teraz[1] := 0;
   najlepsieN := 0;
   skus;
   for i:=1 to najlepsieN do vypis(najlepsie[i]);
end.
```

Riešenia krajského kola kategórie A

A-II-3 Horár Jedlička

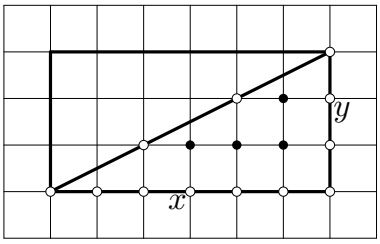
Intuitívne sa zdá, že počet mrežových bodov vo vnútri mnohouholníka bude približne úmerný jeho obsahu. A naozaj, pre všetky mnohouholníky s vrcholmi v mrežových bodoch platí tzv. *Pickova veta*: $S = V + H/2 - 1$, kde S je obsah mnohouholníka, V je počet mrežových bodov vo vnútri mnohouholníka a H je počet mrežových bodov na hranici (vrátane vrcholov). Ukážeme, ako sa dá toto tvrdenie objaviť, a náš postup bude zároveň dôkazom.

Základné pozorovanie, ktoré použijeme, je nasledovné: Predstavte si, že zoberieme mnohouholník a rozstrihneme ho po úsečke, ktorá spája dva mrežové body. Dostaneme takto dva menšie mnohouholníky. Celková plocha zjavne zostala rovnaká. Takisto zostal rovnaký počet mrežových bodov, ktoré oba dokopy obsahujú. Jediné, čo sa zmenilo, je, že keď sme strihali, tak sme možno prestrihli niekoľko mrežových bodov. Tieto doteraz boli vo vnútri jedného mnohouholníka, odteraz sú na obvode dvoch. (To intuitívne zdôvodňuje, prečo sa v Pickovej vete mrežové body na obvode rátajú len za polovicu.)

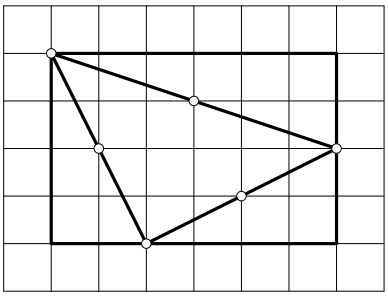
Teraz postupne dokážeme, že Pickova veta platí pre pekné obdĺžniky, pre pravouhlé trojuholníky, pre ľubovoľné trojuholníky a nakoniec pre všeobecné mnohouholníky.

Začneme najjednoduchším špeciálnym prípadom – obdĺžnik, ktorého strany sú rovnobežné so súradnicovými osami. Bez ujmy na všeobecnosti nech sú jeho protiľahlé rohy v bodoch $[0, 0]$ a $[x, y]$. Potom jeho plocha je $S = xy$, mrežových bodov vnútri je $V = (x - 1)(y - 1)$ a na obvode ich je $H = 2x + 2y$. A teda naozaj platí $S = V + H/2 - 1$.

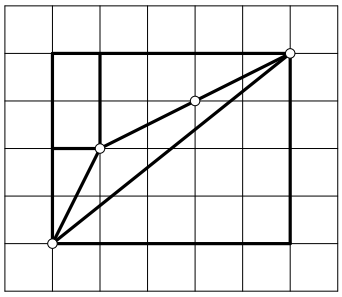
Teraz si predstavme, že takýto obdĺžnik rozstrihneme po uhlopriečke. Dostaneme dva rovnaké pravouhlé trojuholníky:



Obr. 1: Obdĺžnik a dva pravouhlé trojuholníky.



Obr. 2: Vystrihávame všeobecný trojuholník, príklad 1.



Obr. 2: Vystrihávame všeobecný trojuholník, príklad 2.

Označme si x počet mrežových bodov (iných ako rohy obdĺžnika), ktoré ležia na práve prestrihnutej uhlopriečke. Potom každý z trojuholníkov má: plochu $S' = S/2$, mrežových bodov vnútri $V' = (V - x)/2$ a mrežových bodov na obvode $H' = H/2 + x + 1$. A ľahko overíme, že platí $S' = V' + H'/2 - 1$.

Teraz už teda vieme, že Pickova veta platí pre pravouhlé trojuholníky. Čo so všeobecným trojuholníkom? Všeobecnému trojuholníku môžeme opísať obdĺžnik. Dva základné prípady, ktoré môžu nastať, sú znázornené na obrázkoch 2 a 3. Vždy dostaneme približne to isté: Obdĺžnik rozdelený na niekoľko častí. Jedna z nich je trojuholník, ktorý nás zaujíma, a ostatné sú útvary, o ktorých už vieme, že pre ne Pickova veta platí. Rovnakým postupom ako v predchádzajúcom odseku dostaneme, že potom nutne musí platiť aj pre náš trojuholník.

Ostáva nám len posledný krok – všeobecné mnohouholníky. A tento krok je vlastne opäť len zopakovanie tej istej úvahy, len tentokrát naopak. Každý mnohouholník má aspoň jednu *trianguláciu*, t. j. dá sa pozdĺž $N - 3$ vhodných uhlopriečok rozstrihnúť na $N - 2$ neprekrývajúcich sa trojuholníkov.[1] Pre každý z nich platí Pickova veta. A keď zoberieme dva mnohouholníky, o ktorých vieme, že pre ne platí, a zlepíme ich dokopy, bude Pickova veta platiť aj pre nový mnohouholník – na jednej strane rovnice dostaneme súčet plôch, na druhej sa sčítajú mrežové body vnútri a na obvode.

To isté ešte raz a poriadnejšie: Nech mnohouholník M vznikol zlepením mnohouholníkov A a B, so spoločnou stranou na ktorej vnútrajšku leží x mrežových bodov. Označme počet mrežových bodov vnútri A ako V_A a počet mrežových bodov na hranici A ako H_A. Podobne V_B a H_B je počet mrežových bodov vo vnútri

[1] Jeden možný dôkaz vyzerá nasledovne: Pre konvexné mnohouholníky triangluláciu nájdeme ľahko. Ak máme nekonvexný mnohouholník, zoberme jeho jeden nekonvexný vrchol, otočme ho tak, aby obe strany z neho išli „niekam dohora“, zoberme polpriamku z tohto vrcholu „rovno dodola“ a otáčajme ju, kým nenarazí na nejaký vrchol. V tomto okamihu sme našli uhlopriečku, ktorá leží celá vovnútri a môžeme strihať. Indukciou ďalej.

a na hranici B. Potom $V = V_A + V_B + x$ a $H = H_A + H_B - 2x - 2$. Z toho vyplýva, že $(V_A + H_A/2 - 1) + (V_B + H_B/2 - 1) = (V_A + V_B + x) + (H_A + H_B - 2x - 2)/2 - 1 = V + H/2 - 1$. Ak teda obsah A je rovný $V_A + H_A/2 - 1$ a obsah B sa rovná $V_A + H_B/2 - 1$, potom obsah M sa rovná $V + H/2 - 1$.

Trochu iný dôkaz:

Pre zaujímavosť uvedieme myšlienku ešte jedného dôkazu Pickovej vety. Pre každý mrežový bod $[x, y]$ vo vnútri alebo na hranici daného mnohouholníka P označme $\varphi_{P,x,y}$ veľkosť uhla, pod ktorým z neho vidíme vnútro mnohouholníka. Teda ak $[x, y]$ je vnútri, tak $\varphi_{P,x,y} = 2\pi$, ak je na hrane, tak $\varphi_{P,x,y} = \pi$, a pre vrcholy môže nadobúdať rôzne hodnoty.

Sčítajme teraz tieto uhly pre všetky mrežové body v rovine. Presnejšie, definujme $\Psi(P) = \frac{1}{2\pi} \sum \varphi_{P,x,y}$.

Označme teraz, rovnako ako v predchádzajúcom riešení, obsah nášho N-uholníka S_P, počet mrežových bodov v jeho vnútri V_P a počet mrežových bodov na jeho obvode H_P. Zjavne každý z V_P mrežových bodov vnútri prispeje do $\Psi(P)$ jednotkou, každý z $H_P - N$ bodov na hranách jednou polovicou, a všetkých N vrcholov dokopy prispeje $(N-2)/2$ – to preto, že súčet vnútorných uhlov N-uholníka je $2\pi(N-2)$. Dokopy teda $\Psi(P) = V_P + (H_P - N)/2 + (N-2)/2 = V_P + H_P/2 - 1$.

A už zostáva len dokázať, že súčasne platí, že $\Psi(P)$ je zároveň rovné ploche P. To platí napríklad z podobných dôvodov ako v predchádzajúcom riešení – platí to pre trojuholníky, a pre neprekrývajúce sa P, Q zjavne platí $\Psi(P \cup Q) = \Psi(P) + \Psi(Q)$.

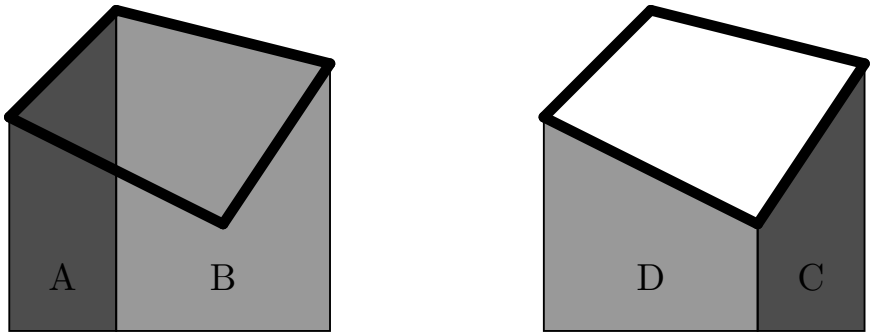
Riešenie úlohy:

Pomocou práve dokázanej vety vieme našu úlohu previesť na dve jednoduchšie: Stačí nám zistiť obsah zadaného mnohouholníka a počet mrežových bodov na jeho hranici.

Počet bodov na hranici vypočítame tak, že sčítame počet vrcholov a pre každú stranu počet mrežových bodov v jej vnútri. Počet mrežových bodov vo vnútri strany s koncami $[x_1, y_1]$ a $[x_2, y_2]$ vypočítame ako „najväčší spoločný deliteľ čísel $|x_2 - x_1|$ a $|y_2 - y_1|$, mínus jedna“.[2] Najväčšieho spoločného deliteľa ľahko spočítame pomocou Euklidovho algoritmu.

[2]Dôkaz: Body na uvedenej úsečke vieme parametricky vyjadriť ako $[x_1 + t(x_2 - x_1), y_1 + t(y_2 - y_1)]$ pre $t \in (0, 1)$. Mrežových bodov je teda toľko ako rôznych hodnôt t, pre ktoré sú $t(x_2 - x_1)$ aj $t(y_2 - y_1)$ celé. Nech d je najväčší spoločný deliteľ $x_2 - x_1$ a $y_2 - y_1$. Potom existujú celé a, b také, že $d = a(x_2 - x_1) + b(y_2 - y_1)$. Ak majú $t(x_2 - x_1)$ aj $t(y_2 - y_1)$ byť celé, musí byť celé číslo aj $at(x_2 - x_1) + bt(y_2 - y_1) = td$. A toto spĺňajú len $t \in \{1/d, 2/d, \ldots, (d-1)/d\}$.

Obsah mnohouholníka ľahko určíme napríklad tak, že pre každú jeho stranu zostrojíme lichobežnik, ktorého jedna strana je strana mnohouholníka, dve strany sú zvislé a štvrtá leží na osi x. Ak máme mnohouholník popísaný proti smeru hodinových ručičiek, dostaneme jeho plochu tak, že sčítame plochy pre lichobežniky, pre ktoré strana mnohouholníka „ide doľava“ a odčítame plochy pre lichobežniky, ktorých strana mnohouholníka „ide doprava“. Napríklad na nasledujúcom obrázku lichobežniky znázornené vľavo pripočítame k ploche a tie znázornené vpravo odpočítame.



Na určenie obsahu lichobežníka použijeme známy vzorec – priemer dĺžok základní $\times$ výška. Vzhľadom k tomu, že tak ako aj v tomto vzorci ako aj vo vzorci na určenie počtu mrežových bodov sa vyskytuje delenie dvoma, budeme radšej počítať dvojnásobok obsahu, aby sme mohli používať celé čísla.

(Alternatívny spôsob počítania obsahu je pomocou vektorového súčinu. Namiesto lichobežníkov rozdelíme mnohouholník na N trojuholníkov, pričom vrcholy i-teho trojuholníka sú $[x_i, y_i]$, $[x_{i+1}, y_{i+1}]$ a $[0, 0]$. Plochu mnohouholníka dostaneme ako súčet orientovaných plôch týchto trojuholníkov, pričom orientovaná plocha takéhoto trojuholníka je polovica z-ovej súradnice vektorového súčinu vektorov $[x_i, y_i, 0]$ a $[x_{i+1}, y_{i+1}, 0]$. Na konci riešenia tejto úlohy uvádzame listing veľmi stručného riešenia v C++ založeného na tejto myšlienke.)

Časová a pamäťová zložitosť:

Časová zložitosť určenia obsahu je lineárna od počtu vrcholov. Časová zložitosť určenia počtu bodov na hranici závisí na zložitosti určenia najväčšieho spoločného deliteľa. Ak použijeme Euklidov algoritmus, tak je táto zložitosť $O(\log \min(X, Y))$, kde X a Y sú obmedzenia na velkosť súradníc vrcholov mnohouholníka. Výsledná časová zložitosť je teda $O(N \log \min(X, Y))$. Okrem súradníc mnohouholníka si potrebujeme pamätať iba konštantné množstvo medzivýsledkov, pamäťová zložitosť je teda $O(N)$.

Listing programu:

```
program les;

const MAXN = 100000;

type bod = record x, y : longint; end;

var n : longint;
    body : array [1 .. MAXN] of bod;
    hranice, dobsah : longint;

procedure nacti_vstup;
var i : longint;
begin
  readln (n);
  for i := 1 to n do readln (body[i].x, body[i].y);
end;

function nsd (n1, n2 : longint) : longint;
var t : longint;
begin
  if n1 > n2 then begin  t := n1;  n1 := n2;  n2 := t;  end;
  while n1 > 0 do begin  t := n1;  n1 := n2 mod n1;  n2 := t;  end;
  nsd := n2;
end;

function pocet_vnitrnich_bodu_usecky (a, b : bod) : longint;
var dx, dy : longint;
begin
  dx := abs (a.x - b.x);
  dy := abs (a.y - b.y);
  if (dx = 0) and (dy = 0) then pocet_vnitrnich_bodu_usecky := 0
    else pocet_vnitrnich_bodu_usecky := nsd (dx, dy) - 1;
end;

function bodu_na_hranici : longint;
var i, b : longint;
begin
  b := n + pocet_vnitrnich_bodu_usecky (body[1], body[n]);
  for i := 1 to n - 1 do
     b := b + pocet_vnitrnich_bodu_usecky (body[i], body[i + 1]);
  bodu_na_hranici := b;
end;

function dvakrat_obsah_lichobeznika (a, b : bod) : longint;
begin
  dvakrat_obsah_lichobeznika := (a.y + b.y) * (a.x - b.x);
end;

function dvakrat_obsah_mnohouhelnika : longint;
```

```
var i, s : longint;
begin
  s := dvakrat_obsah_lichobeznika (body[n], body[1]);
  for i := 1 to n - 1 do
     s := s + dvakrat_obsah_lichobeznika (body[i], body[i + 1]);
  dvakrat_obsah_mnohouhelnika := abs (s);
end;

begin
  nacti_vstup;
  hranice := bodu_na_hranici;
  dobsah := dvakrat_obsah_mnohouhelnika;
  writeln (1 + (dobsah + hranice) div 2);
end.
```

Listing programu:

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int N; cin >> N;
  vector<int> X(N+1), Y(N+1);
  for (int i=0; i<N; i++) cin >> X[i] >> Y[i];
  X[N]=X[0]; Y[N]=Y[0];
  int dvakrat_plocha = 0, na_obvode = 0;
  for (int i=0; i<N; i++) dvakrat_plocha += X[i]*Y[i+1]-X[i+1]*Y[i];
  for (int i=0; i<N; i++) na_obvode += __gcd(abs(X[i+1]-X[i]),abs(Y[i+1]-Y[i]));
  cout << (dvakrat_plocha + na_obvode + 2)/2 << endl;
}
```

A-II-2 Babylonská kríza

Každý si určite všimol podobnosť s úlohou z domáceho kola. Táto podobnosť nie je náhodná, pretože v riešení tejto úlohy použijeme rovnakú dátovú štruktúru ako v riešení úlohy A-I-1. Táto štruktúra sa volá písmenkový strom, alebo tiež trie, a jej popis je vo vzorovom riešení práve spomenutej úlohy domáceho kola.

Zaveďme si označenie, ktoré nám uľahčí ďalší popis. W bude označovať počet wordlistov, L_W počet písmen vo všetkých wordlistoch a L_T dĺžku analyzovaného textu v znakoch.

Asi najjednoduchší spôsob, ktorý každého okamžite napadne, je postupne si pre každý jazyk spraviť samostatný písmenkový strom. Potom stačí prejsť

vstupný text po jednotlivých slovách a pre každé slovo zistiť, v ktorých stromoch (a teda aj wordlistoch) sa nachádza. Každý taký výskyt započítame pre jednotlivé jazyky. Nakoniec nájdeme jazyk s maximálnym počtom takýchto výskytov a jednoducho vypíšeme odpoveď.

Takto priamočiary postup má zložitosť $O(L_T \cdot W + L_W)$, pretože každé slovo v texte vyhľadávame v písmenkovom strome toľko krát, koľko máme písmenkových stromov (a tých je toľko, koľko je wordlistov). Vytvorenie všetkých stromov nedokážeme spraviť lepšie ako v čase L_W, pretože každé slovo z každého wordlistu musí byť pridané do zodpovedajúceho stromu. Pamäťová zložitosť bude prinajhoršom $O(L_W)$.

To síce nie je zlé, ale určite vás napadla kopa zlepšení. Jedným by mohlo byť to, že postavíme spoločný písmenkový strom pre všetky wordlisty a pri každom slove si zapamätáme, do ktorých wordlistov patrí. Toto sa dá implementovať napríklad tak, že v každom vrchole budeme mať zoznam jazykov, ktoré obsahujú slovo zodpovedajúce dotyčnému vrcholu.

Analyzovaný text potom znova prečítame po jednotlivých slovách. Keď nájdeme slovo v strome, nebudeme hneď prechádzať cez všetky jazyky, ktoré ho obsahujú, ale len si v danom vrchole zvýšime počítadlo počtu výskytov dotyčného slova. Až keď dočítame celý text, tak raz celý strom prejdeme (ideálne prehľadávaním do hĺbky) a pri tom spracujeme pre každé slovo, ktoré sa v texte vyskytovalo, všetky jeho výskyty naraz.

Je dôležité si uvedomiť, že vytvorenie takého písmenkového stromu a zároveň prehľadávanie na konci nezaberie viac než $O(L_W)$. Pridané spájané zoznamy totiž celkovú časovú ani pamäťovú zložitosť neovplyvnia. Je to preto, že každý prvok v zozname zastupuje jedno slovo. V strome tak máme najviac $O(L_W)$ normálnych vrcholov a najviac $O(L_W)$ prvkov spájaného zoznamu. Pri vytváraní aj pri záverečnom prehľadávaní tak každý vrchol spracovávame práve raz, takže celková časová zložitosť je $O(L_W + L_T)$. Pamäťová je z rovnakých príčin $O(L_W)$.

Lepšiu asymptotickú časovú zložitosť nevieme dosiahnuť, pretože vstupné súbory musíme aspoň raz prečítať.

Aj keď vieme, že ďalšie zrýchlenie sa už nedá dosiahnuť, môžeme sa pokúsiť o implementačne jednoduchšie riešenie. V práve popísanom algoritme sme si pre každé slovo pamätali, v ktorých wordlistoch sa nachádza. Skúsme túto informáciu zo stromu úplne vynechať.

Po jednom prečítaní vstupného textu budeme o každom slove vedieť, koľkokrát sa v texte vyskytuje. Nevieme len, v ktorých wordlistoch sa tieto slová nachádzajú. Nič nám ale nebráni v tom, aby sme si prečítali všetky wordlisty

ešte raz. Keď teraz čítame wordlist, každé jeho slovo v strome nájdeme a zodpovedajúcemu jazyku prirátame jeho počet výskytov v texte.

Takéto prejdenie bude trvať $O(L_W)$ krokov, takže to celkovú časovú zložitosť nepokazí. Navyše však bude tento algoritmus omnoho jednoduchší na implementáciu.

Listing programu:

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

#define WORD_LEN 255
#define ALPHABET_SIZE 26

struct TRIE_NODE {
      TRIE_NODE *next[ALPHABET_SIZE];
      int count;
   };

struct DICT_DESC {
      char name[WORD_LEN];
      int size;
   };

void trie_add(TRIE_NODE *root, const char *word)
{
   TRIE_NODE *cur_root = root;

   while (*word)
   {
      if (!cur_root->next[*word-'a'])
      {
         cur_root->next[*word-'a'] = malloc(sizeof(TRIE_NODE));
         for (int i=0; i<ALPHABET_SIZE; i++)
            cur_root->next[*word-'a']->next[i] = NULL;
         cur_root->next[*word-'a']->count = 0;
      }

      cur_root = cur_root->next[*word-'a'];
      ++word;
   }
   //Nie je potrebne pamatat si konce slov,
   //v zaverecnom pocitani sa hodnoty vo vrcholoch zanedbaju
}

void trie_increase_word_count(TRIE_NODE *root, const char *word)
{
   TRIE_NODE *cur_root = root;    //posledna faza, mame istotu, ze slovo najdeme
   while (*word && cur_root)
```

```
   {
      cur_root = cur_root->next[*word-'a'];
      ++word;
   }
   if (cur_root)
      cur_root->count++;
}

int trie_find(TRIE_NODE *root, const char *word)
//najdeme slovo v strome vratime, ze kolko krat sa vyskytlo
{
   TRIE_NODE *cur_root = root;    //posledna faza, mame istotu, ze slovo najdeme

   while (*word)
   {
      cur_root = cur_root->next[*word-'a'];
      ++word;
   }
   return cur_root->count;
}

void trie_free(TRIE_NODE *root)  //uvolnenie struktur
{
   if (!root)
      return;
   for(int i=0; i<ALPHABET_SIZE; i++)
      trie_free(root->next[i]);
   free(root);
}

TRIE_NODE *trie_root;
int N;

int main(void)
{
   FILE *fdict = fopen("slovniky.in", "r");
   FILE *ftext = fopen("text.in", "r");

   trie_root = malloc(sizeof(TRIE_NODE));
   for (int i=0; i<ALPHABET_SIZE; ++i)
      trie_root->next[i] = NULL;
   trie_root->count = 0;

   //inicializacia pismenkoveho stromu
   int dict_count;
   fscanf(fdict, "%d", &dict_count);
   for (int dict=0; dict<dict_count; dict++)
   {
      int dict_size;
      char word[WORD_LEN+1];
```

```
   fscanf(fdict, "%s %d", word, &dict_size);
   for(int cur_word=0; cur_word<dict_size; cur_word++)
   {
      fscanf(fdict, "%s", word);
      trie_add(trie_root, word);
   }
}

//zistenie poctu slov v texte
while( !feof(ftext))
{
   char word[WORD_LEN+1];
   fscanf(ftext, " %s ", word);
   trie_increase_word_count(trie_root, word);
}

//zistenie poctu slov v slovniku
fclose(fdict);
fdict = fopen("slovniky.in", "r");
fscanf(fdict, "%d", &dict_count);
DICT_DESC *dict_desc = malloc(sizeof(DICT_DESC) * dict_count);

for (int dict=0; dict<dict_count; dict++)
{
   int dict_size;
   fscanf(fdict, "%s %d", dict_desc[dict].name, &dict_size);
   dict_desc[dict].size = 0;

   for(int cur_word=0; cur_word<dict_size; cur_word++)
   {
      char word[WORD_LEN+1];
      fscanf(fdict, "%s", word);
      dict_desc[dict].size+= trie_find(trie_root, word);
   }
}

//najdenie slovniku s najvacsim poctom slov
int max = 0;
for (int dict=0; dict<dict_count; dict++)
   if (dict_desc[dict].size>max)
      max = dict_desc[dict].size;

//vypisanie slovnikov
FILE *fout = fopen("jazyk.out", "w");
for (int dict=0; dict<dict_count; dict++)
   if (dict_desc[dict].size==max)
      fprintf(fout, "%s\n", dict_desc[dict].name);
fclose(fout);

//uvolnenie stromu
```

```
    trie_free(trie_root);
    return 0;
}
```

A-II-3 Cyklistické preteky

Hneď po prečítaní zadania je jasné, že táto úloha bude mať niečo spoločné s grafovými algoritmami. Mestá budú vrcholy, možné etapy pretekov budú hrany a počty divákov si môžeme predstaviť ako ohodnotenie hrán nezápornými celými číslami. Ako ale nájdeme riešenie?

Začneme prvou časťou úlohy, teda nájdením nejakej trasy pretekov z Vyšných Hákov do Veľkého Sumca s čo najmenej etapami (ale bez požiadavky na maximalizáciu počtu divákov). Takto zadaná úloha je v podstate učebnicovým príkladom na prehľadávanie grafu do šírky. Pre každé mesto si spočítame minimálny počet etáp nutných na jeho dosiahnutie a to tak, že Vyšné Háky dostanú nulu, ich susedia jednotku, susedia susedov dvojku atď. V grafe nám tak vzniknú akési vrstvy, pričom v k-tej vrstve sú vrcholy, do nich sa dá dostať najkratšou cestou na k etáp. Nájdenie nejakej trasy je potom jednoduché: Začneme z posledného vrcholu, teda z Veľkého Sumca, potom zoberieme nejakého jeho suseda vo vrstve o jedna nižšej a tak pokračujeme, pokiaľ sa nedostaneme do Vyšných Hákov. Toto riešenie vyžaduje lineárny čas a lineárne množstvo pamäte, tj. $O(N + M)$.

A ako nájdeme tú divácky najatraktívnejšiu trasu? Pre každé mesto už vieme najmenší počet etáp, na ktorý sa doň vieme dostať. Teraz navyše pre každé mesto spočítame, koľko najviac divákov môže vidieť niektorú najkratšiu trasu z Vyšných Hákov do tohto mesta.

Postup, ako tento údaj spočítame, bude podobný tomu z predchádzajúcich odsekov. Pre mestá susediace s Vyšnými Hákmi je to priamo počet divákov na etape medzi nimi (keďže musíme použiť najmenší počet etáp, nemáme na výber). Potom postupne spracujeme mestá, kam sa vieme dostať na 2 etapy, potom na 3, atď. Keď teraz zisťujeme optimálny počet divákov pre konkrétne mesto X vo vzdialenosti k etáp od Vyšných Hákov, nájdeme si všetkých jeho susedov $Y_1, \ldots, Y_l$, do ktorých sa z Vyšných Hákov dalo dostať na $k - 1$ etáp. Najlepšia cesta do X určite vyzerá tak, že na $k - 1$ etáp prídeme (optimálnym spôsobom, ktorý už v tomto okamihu poznáme) do niektorého mesta Y_i, a následne z neho k-tou etapou do X. Z týchto l možností si vyberieme tú najlepšiu.

Zjavne každý vrchol spracujeme práve raz a každú hranu práve dvakrát, preto je časová aj pamäťová zložitosť lineárna od veľkosti daného grafu – teda $O(M + N)$. (Aby sme dosiahli túto časovú a pamäťovú zložitosť, graf máme uložený ako zoznam okolí vrcholov – pre každý vrchol si pamätáme zoznam hrán, ktoré z neho vedú.)

Za zmienku ešte stojí malý trik, ako program rieši výpis trasy – namiesto hľadania trasy od konca a následného otočenia poradia miest, ako by naznačoval algoritmus, nájdeme cestu z Veľkého Sumca do Vyšných Hákov a pri hľadaní trasy od konca ju rovno vypisujeme – takže vypíšeme trasu z Vyšných Hákov do Veľkého Sumca.

Na konci uvádzame aj alternatívne riešenie v C++, ktoré obe časti spája do jednej a počas prehľadávania do šírky rovno spočíta aj optimálny počet divákov.

Listing programu:

```
program zavod;

const maxN = 1000000;
      maxM = 1000000;
      zacatek = 2;  { prohodili jsme startovni a cilove mesto, abychom }
      konec = 1;    { nemuseli otacet cestu }

var M,N:integer;

    mesta: array [1..maxN] of record { informace o mestech }
            vzdalenost : integer;   { pocet etap od zacatku }
            maxDivaku : integer;    { pocet divaku, ktery shledne trasu az sem}
            predchozi:integer;      { predchozi mesto na nejlepsi trase }
            stupen: integer;        { pocet sousednich mest}
            zacatek: integer;       { pozice prvniho souseda v poli sousede }
            pozice:integer; {docasna hodnota pouzivana pri nacitani}
        end;
    sousede: array[1..2*maxM] of record {pole sousedu. nejprve jsou ulozeni }
                { sousede mesta 1, pak sousede mesta 2, atd. }
            mesto, divaku:integer;
        end;
    hrany: array[1..maxM] of record   { pole hran, neboli moznych etap. Pouzito}
                                      { jen pri nacitani. }
        odkud, kam, divaku: integer;  { odkud a kam etapa vede a pocet divaku }
        end;

{ Nacteni dat ze vstupu - naplni pole mesta a sousede. }
procedure nacti;
var i, pozice:integer;
    a,b, divaku:integer; {mesta}
    p:integer;
begin
```

```
    readln(N, M);
    for i := 1 to N do begin
        mesta[i].stupen := 0;
    end;
    {nacti hrany a spocti stupne}
    for i:= 1 to M do begin
        readln(a, b, hrany[i].divaku);
        hrany[i].odkud := a; hrany[i].kam := b;
        mesta[a].stupen := mesta[a].stupen + 1;
        mesta[b].stupen := mesta[b].stupen + 1;
    end;
    { urci zacatky pro jednotliva mesta v poli sousede }
    pozice := 1; {prubezna pozice v poli sousede}
    for i := 1 to N do begin
        mesta[i].zacatek := pozice;
        mesta[i].pozice := pozice;
        pozice := pozice + mesta[i].stupen;
    end;
    { napln pole sousede }
    for i:= 1 to M do begin
        a := hrany[i].odkud; b := hrany[i].kam; divaku := hrany[i].divaku;
        {pridej souseda mestu a}
        p:= mesta[a].pozice;
        sousede[p].mesto := b;
        sousede[p].divaku := divaku;
        mesta[a].pozice := p+ 1;
        {pridej souseda mestu b}
        p:= mesta[b].pozice;
        sousede[p].mesto := a;
        sousede[p].divaku := divaku;
        mesta[b].pozice := p+ 1;
    end;

end;

{ Fronta mest ke zpracovani. Vzhledem ke zpusobu pridavani plati, ze }
{ mesto s nizsi vzdalenosti od zacatku je v poli na nizsi pozici nez }
{ mesto s vetsi vzdalenosti. }
var fronta: array[1..maxN] of integer;
    zacF,konF:integer; {zacatek a konec fronty}

{ Projde sousedy vybraneho mesta a pokusi se prodlouzit do nich trasu. }
procedure projdiSousedy(mesto:integer);
var i: integer; {index do pole sousedu}
    posledni: integer; {index posledniho souseda v poli sousede. }
    divaku, soused : integer;
begin
    posledni := mesta[mesto].zacatek + mesta[mesto].stupen - 1;
    for i:= mesta[mesto].zacatek to posledni do begin
        divaku := mesta[mesto].maxDivaku + sousede[i].divaku;
```

```
        soused := sousede[i].mesto;
        if mesta[soused].vzdalenost = -1 then begin
            {mesto jsme jeste nenavstivili.}
            konF := konF + 1;
            fronta[konF] := soused; {dej do fronty}
            mesta[soused].vzdalenost := mesta[mesto].vzdalenost + 1;
            mesta[soused].maxDivaku := divaku; { a toto je zatim nejlepsi trasa}
            mesta[soused].predchozi := mesto;
        end else if (mesta[soused].vzdalenost = mesta[mesto].vzdalenost + 1)
            and (mesta[soused].maxDivaku < divaku) then begin
            { trasa pses toto mesto je lepsi, prenastavime hodnoty u souseda.}
            mesta[soused].maxDivaku := divaku;
            mesta[soused].predchozi := mesto;
        end;
    end;
end;

{ Spocte hodnoty maxDivaku, vzdalenost a predchozi v poli mesta, cimz }
{ prakticky vyresi celou ulohu. }
procedure spocti;
var mesto: integer;
    i:integer;
begin
    {inicializace}
    for i := 1 to N do begin
        mesta[i].maxDivaku := 0;
        mesta[i].vzdalenost := -1; {zatim jsme nikde nebyli}
    end;
    {pocatecni mesto ma vzdalenost 0 a zadne divaky}
    mesta[zacatek].vzdalenost := 0;
    mesta[zacatek].maxDivaku := 0;
    fronta[1] := zacatek;
    zacF := 1; konF := 1;
    {dokud neni fronta prazdna}
    while zacF <= konF do begin
        mesto := fronta[zacF];
        zacF := zacF + 1;  {precti hodnotu z fronty}
        projdiSousedy(mesto);
    end;
end;

{ Vypise cestu od konce do zacatku. Protoze cesta, kterou najde spocti(), }
{ konci ve Vysnych Hacich, tak je vysledkem presne to, co pozaduje zadani. }
procedure vypis;
var mesto:integer;
begin
    writeln(mesta[konec].vzdalenost, ' ', mesta[konec].maxDivaku);
    mesto := konec;
    write(konec);
    while mesto <> zacatek do begin   {dokud nejsme na zacatku}
```

```
        mesto := mesta[mesto].predchozi; {najdi mesto, ze ktereho jsme prisli}
        write(' ', mesto);       {vypis ho}
    end;
    writeln;
end;


{ Hlavni program }
begin
    nacti;
    spocti;
    vypis;
end.
```

Listing programu:

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
struct hrana { public: int ciel, pocet; };

int main() {
  int N, M; cin >> N >> M;
  vector< vector< hrana > > G(N);
  // nacitaj vstup
  for (int i=0; i<M; i++) {
    int x; hrana E;
    cin >> x >> E.ciel >> E.pocet;
    x--; E.ciel--;
    G[x].push_back(E);
    swap(x,E.ciel);
    G[x].push_back(E);
  }
  // prehladaj do sirky
  vector<int> vzdialenost(N,987654321), divakov(N), odkial(N);
  queue<int> Q;
  Q.push(0); vzdialenost[0]=divakov[0]=odkial[0]=0;
  while (!Q.empty()) {
    int kde = Q.front(); Q.pop();
    for (int i=0; i<int(G[kde].size()); i++) {
      int kam = G[kde][i].ciel, kolko = G[kde][i].pocet;
      if (vzdialenost[kam] > vzdialenost[kde]+1) {
        vzdialenost[kam] = vzdialenost[kde]+1;
        divakov[kam] = divakov[kde] + kolko;
        odkial[kam] = kde;
        Q.push(kam);
      }
```

```
      if (vzdialenost[kam] == vzdialenost[kde]+1)
        if (divakov[kam] < divakov[kde] + kolko) {
          divakov[kam] = divakov[kde] + kolko;
          odkial[kam] = kde;
        }
    }
  }
  // vypis vzdialenost a pocet divakov
  cout << vzdialenost[1] << " " << divakov[1] << endl;
  // zostroj a vypis cestu
  vector<int> cesta(1,1);
  while (cesta.back()!=0) cesta.push_back(odkial[cesta.back()]);
  reverse(cesta.begin(),cesta.end());
  for (int i=0; i<int(cesta.size()); i++)
    cout << (i?" ":"") << (cesta[i]+1);
  cout << endl;
}
```

A-II-4 Zásobníkové počítače

Vzorové riešenie používa len jeden zásobník. V ňom si budeme počas výpočtu udržiavať čo najjednoduchší výraz, ktorý je ekvivalentný s doteraz prečítanou časťou vstupu.

Zabudnime na chvíľku na zátvorky (budeme predpokladať, že nie sú na vstupe). Výraz budeme načítavať po znakoch. Ak načítame hodnotu `0` alebo `1`, uložíme ju na vrch zásobníka. Ak načítame nejaký operátor, tak výraz v zásobníku trošku zjednodušíme a potom načítaný operátor vložíme na vrch zásobníka. V zásobníku sa nám budú striedať operátory a hodnoty, pričom na spodku zásobníka bude hodnota.

Zjednodušovanie výrazov: Ak načítame operátor `|`, tak vtedy môžeme celý výraz, ktorý je uložený v zásobníku vyhodnotiť. Vieme to spraviť napríklad tak, že vyberieme vrchné 3 prvky zo zásobníka, ktoré budú mimochodom vyzerať ako `hodnota, operátor, hodnota`, spracujeme[3] a výslednú hodnotu vložíme na vrch zásobníka. Toto budeme opakovať až kým nebude zásobník obsahovať iba jednu hodnotu. To môžeme považovať za dostatočne zjednodušený výraz.

Všimnite si, že ak by v zásobníku boli za sebou (oddelené číslom) dva operátory `&` a nad nimi operátor `|`, tak by sme týmto postupom mohli prísť k zlému výsledku[4]. Preto treba zaručiť, aby sa za sebou nenachádzalo dva a viac

[3] Vyhodnotíme výraz, ktorý reprezentujú

[4] Skúste si nájsť taký príklad.

operátorov `&`. Kvôli tomu, vždy ak načítame operátor `&`, tak predtým než ho vložíme na vrch zásobníka skontrolujeme, či by neboli za sebou dva operátory `&` (vtedy zásobník vyzerá ako `hodnota, &, hodnota`). Ak hej, tak danú trojicu vyberieme zo zásobníka, vyhodnotíme, a výsledok vložíme na vrch zásobníka. Toto budeme opakovať až kým nebude zásobník obsahovať iba jednu hodnotu, alebo nenarazíme na operátor `|`.

Zátvorky: Teraz sa musíme popasovať so zátvorkami. Ak načítame ľavú zátvorku, musíme rekurzívne vyhodnotiť výraz medzi touto zátvorkou a správnou pravou zátvorkou. Zásobník nám pri tom veľmi pomôže. Jednoducho vložíme zátvorku do zásobníka a budeme pokračovať vo vyhodnocovaní výrazov tak, ako sme popísali vyššie (jediný rozdiel je, že pri zjednodušovaní výrazov budeme považovať za prázdny zásobník aj zásobník, na ktorý má na vrchu ľavú zátvorku).

Keď načítame pravú zátvorku, tak vyhodnotíme výraz na zásobníku až po najbližšiu ľavú zátvorku (tak, ako pri zjednodušovaní výrazu pri načítaní `|`) a výsledok vložíme do zásobníka (čiže sme nahradili celú zátvorku jej hodnotou).

Keď dočítame vstup, tak nám ostane v zásobníku nejaký výraz, ktorý stačí vyhodnotiť presne tak, ako pri načítaní operátora `|`. Potom ostane v zásobníku práve jedna hodnota, ktorá sa rovná hodnote výrazu zo vstupu.

Tento algoritmus zjavne používa iba jeden zásobník. Časová zložitosť je trosku zložitejšia. Všimnite si, že každý operátor zo vstupu dáme najviac jeden krát do zásobníka a najviac jeden krát ho potom vyberieme. Medzi vloženiami a výbermi vždy spravíme konštantný počet operácii a preto je časová zložitosť lineárna.

Listing programu:

```
program vyraz;
var s : stack of char;
    c : char;

{ Vrati operator na vrchu zasobnika. }
{ Ak tam nie je ziadny, vrati '#'. }
function vrchny_op : char;
var num, op : char;
begin
   num := pop(s);
   if empty(s) then op := '#'
               else begin op := pop(s); push(s, op); end;
   push(s, num);
   vrchny_op := op;
end;
```

```
{ Vyhodnoti operator na vrchu zasobnika. }
procedure vyhodnot;
var a, b, op, vysledok : char;
begin
   b := pop(s);
   op := pop(s);
   a := pop(s);
   if op = '&' then vysledok := a='1' and b='1';
   if op = '|' then vysledok := a='1'  or b='1';
   if op = '(' then vysledok :=           b='1';
   if vysledok then push(s, '1') else push(s, '0');
end;

begin
   while read(c) do case c of
     '0' ,
     '1' : push(s, c);        { Cislo ulozime na zasobnik. }
     '&' : begin              { Vyhodnotit vsetky '&'. }
              while vrchny_op = '&' do vyhodnot;
              push(s, c);
           end;
     '|' : begin              { Vyhodnotit vsetky '&' alebo '|'. }
              while vrchny_op in ['&', '|'] do vyhodnot;
              push(s, c);
           end;
     '(' : begin              { Pred zatvorkou ulozime pomocnu nulu. }
             push(s, '0');
             push(s, c);
           end;
     ')' : begin              { Vyhodnotit az do '('. }
              while vrchny_op <> '(' do vyhodnot;
              vyhodnot;       { Odstrani '(' z vrchu zasobniku. }
           end;
   end;
   while vrchny_op <> '#' do vyhodnot;
   write(pop(s));
end.
```

Riešenia krajského kola kategórie B

B-II-1 Binárny súčet

Ukážeme dve rôzne, rovnako dobré riešenia zadanej úlohy.

Začneme od konca:

Postupne budeme hľadať riešenie hlavolamu začínajúc najmenej významnou cifrou.

Niekedy rovno zistíme, že riešenie neexistuje (napr. pre posledné cifry `0+1=0`). Niekedy bude jediná možnosť, ako posledné cifry doplniť (napr. `0+*=0` alebo `1+*=0`). Občas budú tie možnosti dve, ale navzájom ekvivalentné (napr. `*+*=1` môžeme doplniť ako `1+0=1` aj ako `0+1=1` a je jedno, ktorú možnosť si vyberieme).

Lenže môže nastať aj situácia, v ktorej sa nebudeme priamo vedieť rozhodnúť, ako chýbajúce cifry doplniť: `*+*=0`. Tu totiž máme dve možnosti (`0+0=0` a `1+1=0`), ktoré ale nie sú ekvivalentné – pri druhej z nich nastáva prenos do vyššieho rádu. V tomto okamihu ešte nevieme povedať, ktorú z týchto možností si máme vybrať. Napr. ak by bolo celé zadanie `1*+0*=10`, treba si vybrať prvú možnosť, ale ak by bolo zadanie `01*+00*=100`, tak druhú.

(A niekedy sú dokonca oba výbery dobré, ako napr. pre zadanie `01*+0**=100`. Ale tým sa príliš trápiť nemusíme, keďže nám stačí nájsť jedno riešenie.)

Ako si s tým poradiť? Jeden možný prístup by bol skúsiť postupne obe možnosti doplnenia. Lenže takýto postup vedie k veľmi pomalému riešeniu. Napríklad pre `0*0*0*0*0* + 0********* = 1110101010` by sme takto vyskúšali 2^4 možností doplnenia, z ktorých by ani jedna nefungovala, lebo najvyššie cifry doplniť nevieme. Určite ľahko upravíte tento vstup na taký, pre ktoré by riešenie skúšajúce všetky možnosti potrebovalo prezrieť tých možností 2^{100}.

Omnoho lepšie riešenie dostaneme tak, že si jednoducho zapamätáme, že vieme najpravejšiu cifru doplniť dvoma spôsobmi – aj tak, aby prenos vznikol, aj tak, aby nevznikol.

A to už je vlastne celé riešenie. Budeme spracúvať zadané reťazce sprava doľava a pre každé k zistíme odpoveď na otázky: „Vieme pravých k stĺpcov korektne vyplniť?“ a „Vieme pravých k stĺpcov korektne vyplniť s tým, že vznikne prenos do vyššieho rádu?“. Keď spracúvam stĺpec $k+1$, stačí mi vedieť, ktoré možnosti pre k stĺpcov viem dosiahnuť, a vyskúšať všetky možnosti pre každú hviezdičku v ňom.

V nasledujúcom listingu programu počítame hodnoty `viem[k][x]` – „Viem vyplniť posledných `k` cifier tak, aby všetko sedelo a prenos bol `x`?“. Vždy, keď sa nám podarí nájsť jednu možnosť, ako to dosiahnuť, zapamätáme si v poli `ako` na pozíciách `ako[k][x][0..2]` jednu možnosť, aká musí byť cifra v reťazci A, cifra v B a prenos z rádu $k-1$.

Ak po spočítaní všetkých hodnôt v poli `viem` zistíme, že sa úloha dá riešiť, pomocou hodnôt v poli `ako` jedno riešenie spätným prechodom (zľava doprava) zostrojíme.

Pre každú cifru vyskúšame nanajvýš 8 možností, preto je časová aj pamäťová zložitosť lineárna od počtu cifier v zadaných číslach.

Listing programu:

```
var
   A, B, C : ansistring;
   N, k, lop, hip, loa, hia, lob, hib, prenos, ha, hb, hc, d1, d2 : longint;
   viem : array[0..1000047,0..1] of boolean;
   ako : array[0..1000047,0..1,0..2] of longint;

begin
   readln(A); readln(B); readln(C); N := length(A);
   viem[0][0]:=true; viem[0][1]:=false;
   for k := 1 to N do begin
      lop:=0; hip:=1; loa:=0; hia:=1; lob:=0; hib:=1;
      if not viem[k-1][0] then lop:=1;
      if not viem[k-1][1] then hip:=0;
      if A[N+1-k]<>'*' then begin loa:=ord(A[N+1-k])-48; hia:=loa; end;
      if B[N+1-k]<>'*' then begin lob:=ord(B[N+1-k])-48; hib:=lob; end;
      hc := ord(C[N+1-k])-48;
      viem[k][0]:=false; viem[k][1]:=false;
      for prenos := lop to hip do
         for ha := loa to hia do
            for hb := lob to hib do begin
               d1 := (prenos + ha + hb) mod 2;
               d2 := (prenos + ha + hb) div 2;
               if d1=hc then begin
                  viem[k][d2]:=true;
                  ako[k][d2][0]:=prenos;
                  ako[k][d2][1]:=ha;
                  ako[k][d2][2]:=hb;
               end;
            end;
   end;
   if viem[N][0] then begin
      prenos:=0;
      for k:=N downto 1 do begin
         A[N+1-k]:=chr(48+ako[k][prenos][1]);
```

```
        B[N+1-k]:=chr(48+ako[k][prenos][2]);
        prenos := ako[k][prenos][0];
      end;
      writeln(A); writeln(B);
   end else writeln('Nema riesenie.');
end.
```

Začneme od núl:

Predstavme si, že by sme namiesto všetkých `*` napísali navzájom rôzne premenné. Pre príklad zo zadania dostávame rovnosť `010ab0 + 01cd1e = 101001`. Preložené do matematickej reči, má platiť:

$$(2^4 + a2^2 + b2^1) + (2^4 + c2^3 + d2^2 + 2^1 + e2^0) = (2^5 + 2^3 + 2^0)$$

„Zozbierajme“ si na jednej strane členy obsahujúce neznáme, na druhej zvyšok:

$$c2^3 + a2^2 + d2^2 + b2^1 + e2^0 = (2^5 + 2^3 + 2^0) - (2^4) - (2^4 + 2^1)$$

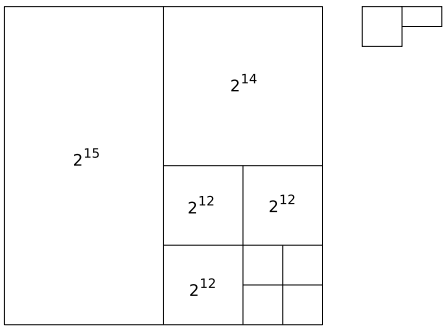
Všimnime si, čo sme na pravej strane dostali – hodnotu $C - A' - B'$, kde A' a B' vzniknú z A a B tak, že za všetky `*` dosadíme nuly.

Nech by zadanie vyzeralo akokoľvek, ľavá strana tejto rovnosti vždy bude nezáporná. Ak nám pravá strana vyjde záporná, môžeme prehlásiť, že úloha nemá riešenie – nech by sme chýbajúce cifry doplnili ako len chceme, vždy dostaneme priveľký výsledok.

Teraz ukážeme, že v opačnom prípade stačí premenné dopĺňať „od najväčšej po najmenšiu“ s tým, že vždy, keď môžeme, priradíme premennej hodnotu 1.

Začneme dôkazom jedného intuitívne zjavného tvrdenia.

Lema. Nech $a_1, \ldots, a_n$ sú nezáporné celé čísla menšie alebo rovné k a nech $2^{a_1} + \cdots + 2^{a_n} \geq 2^k$. Potom sa dá na ľavej strane niektoré sčítance vynechať tak, aby sme dostali súčet presne 2^k.



Dôkaz. Bez ujmy na všeobecnosti nech $a_1 \geq a_2 \geq \cdots$. Nech q je najmenší index taký, že $2^{a_1} + \cdots + 2^{a_q} \geq 2^k$. Potom $2^{a_1} + \cdots + 2^{a_{q-1}} < 2^k$. Všetky sčítance sú deliteľné 2^{a_q}, preto je aj ich súčet deliteľný 2^{a_q}, a teda $2^{a_1} + \cdots + 2^{a_{q-1}} \leq 2^k - 2^{a_q}$. Lenže potom nutne $2^{a_1} + \cdots + 2^{a_q} \leq 2^k$, a teda $2^{a_1} + \cdots + 2^{a_q} = 2^k$.

Na obrázku vpravo je graficky znázornená myšlienka tvrdenia a dôkazu pre $k = 16$ a $a_1, a_2, \ldots = 15, 14, 12, 12, 12, 10, 10, 10, 10, 10, 9$.

Čo nám práve dokázaná lema hovorí o našej úlohe? Nás zaujíma, či existuje riešenie rovnice $x_1 2^{a_1} + \cdots + x_n 2^{a_n} = Q$, kde $x_i \in \{0, 1\}$.

Nech a_1 je najväčšie zo všetkých a_i. Sú dve možnosti: Ak $2^{a_1} > Q$, zjavne musí byť $x_1 = 0$, lebo by sme dostali priveľkú ľavú stranu. A práve dokázaná lema nám hovorí, že v opačnom prípade nič nepokazíme, ak vezmeme $x_1 = 1$. Prečo? Predstavme si, že existuje riešenie zadanej rovnice, v ktorom $x_1 = 0$. Do tohto riešenia sme vybrali niekoľko iných mocnín dvojky, ktorých celkový súčet je $Q \geq 2^{a_1}$. No a podľa lemy vieme vybrať niekoľko z nich tak, aby dali súčet presne 2^{a_1}. Lenže potom vieme zostrojiť nové riešenie, v ktorom všetky tieto mocniny dvoch vyhodíme (nastavíme ich x_i na 0) a namiesto nich použijeme 2^{a_1} (nastavíme x_1 na 1).

Inými slovami, práve sme dokázali: AK ($Q \geq 2^{a_1}$ a existuje nejaké riešenie), TAK (existuje riešenie, kde $x_1 = 1$). A tým sme dokázali, že naozaj funguje „pažravý“ postup, pri ktorom budeme neznáme dopĺňať zľava doprava použitím pravidla „ak môžeš, daj 1, ak nie, daj 0“.

Zostáva upresniť, ako to celé šikovne implementovať. Číslo na pravej strane rovnosti budeme celý čas reprezentovať ako reťazec bitov (nebudeme ho teda vyčísľovať). Rozdiel $Q = C - A' - B'$ vieme spočítať klasickým „druhá-trieda-ZŠ“ postupom v čase lineárnom od dĺžky reťazcov. Takisto si v lineárnom čase vieme zozbierať všetky mocniny dvoch, na ktorých máme v zadaní hviezdičky. No a teraz budeme postupne prechádzať zozbierané mocniny a pre každú rozhodneme, či tam musí byť 0 (a nič sa nedeje), alebo môže byť 1 (a potrebujeme zmenšiť Q).

Dalo by sa ukázať, že už tento algoritmus bude dokopy lineárny od dĺžky zadaných reťazcov, môžeme však spraviť jedno vylepšenie, po ktorom to už bude skutočne očividné.

Keďže na každej pozícii máme najviac dve hviezdičky, čokoľvek, čo dostaneme doplnením za všetky `*` na pozíciách 2^0 až 2^{k-1}, bude menšie ako 2^{k+1}. Ak teda po spracovaní pozície k máme $Q \geq 2^{k+1}$, vieme, že riešenie neexistuje a môžeme prestať. Preto pri každom zmenšení Q budeme meniť najviac dve binárne cifry Q, a teda bude celé toto riešenie lineárne.

(Na záver upozornenie: Toto riešenie naozaj musíme robiť postupne, najskôr spočítať $C - A' - B'$ a až potom dopĺňať hviezdičky. Ak by sme sa „pažravým“ spôsobom pokúšali doplniť hviezdičky priamo, zlyhalo by to napr. pre vstup `0**** + 00**1 = 01000`.)

Listing programu:

```
var
   A, B, C : ansistring;
   Q : array[1..1000047] of longint;
   N, i, j, qtop : longint;

begin
   readln(A); readln(B); readln(C); N := length(A);

   for i := 1 to N do Q[i] := ord(C[N+1-i])-48;
   for i := 1 to N do begin
      if Q[i]<0 then begin dec(Q[i+1]); inc(Q[i],2); end;
      if B[N+1-i]='1' then j:=1 else j:=0;
      if j <= Q[i] then dec(Q[i],j)
                   else begin dec(Q[i+1]); inc(Q[i],2); dec(Q[i],j); end;
   end;
   if Q[N+1]<0 then begin writeln('Nema riesenie.'); halt; end;
   for i := 1 to N do begin
      if Q[i]<0 then begin dec(Q[i+1]); inc(Q[i],2); end;
      if A[N+1-i]='1' then j:=1 else j:=0;
      if j <= Q[i] then dec(Q[i],j)
                   else begin dec(Q[i+1]); inc(Q[i],2); dec(Q[i],j); end;
   end;
   if Q[N+1]<0 then begin writeln('Nema riesenie.'); halt; end;
   qtop:=N; while (qtop>0) and (Q[qtop]=0) do dec(qtop);

   for i := N downto 1 do begin
      if qtop > i+1 then begin writeln('Nema riesenie.'); halt; end;
      if A[N+1-i]='*' then begin
         if qtop < i then A[N+1-i]:='0' else begin
            A[N+1-i]:='1';
            dec(Q[i]); j:=i;
            while Q[j]<0 do begin inc(Q[j],2); dec(Q[j+1]); inc(j); end;
            while (qtop>0) and (Q[qtop]=0) do dec(qtop);
         end;
      end;
      if B[N+1-i]='*' then begin
         if qtop < i then B[N+1-i]:='0' else begin
            B[N+1-i]:='1';
            dec(Q[i]); j:=i;
            while Q[j]<0 do begin inc(Q[j],2); dec(Q[j+1]); inc(j); end;
            while (qtop>0) and (Q[qtop]=0) do dec(qtop);
         end;
      end;
   end;
   if qtop>0 then writeln('Nema riesenie.')
             else begin writeln(A); writeln(B); end;
end.
```

B-II-2 Výber dovolenky

Najprv sa zamyslime nad menej efektívnymi riešeniami. Môžeme skúšať všetky možné podpostupnosti a pre ne zrátať počet jednotlivých písmen. Ak je z každého písmena aspoň po jednom, tak postupnosť je dobrá. Možných začiatkov a koncov je zhruba N^2. Vyrátanie počtu písmen vyžaduje toľko operácií, aká je dlhá podpostupnosť, teda najviac N operácií. Dokopy dostávame časovú zložitosť $O(N^3)$.

Tento postup vieme jednoducho zrýchliť. Predstavme si, že sme vyrátali početnosť písmen pre podpostupnosť od i po j. Aby sme vyrátali početnosť pre podpostupnosť od i po $j+1$, nemusíme ju zbytočne prechádzať celú. Stačí zobrať početnosti pre podpostupnosť od i po j a zvýšiť početnosť písmena na pozícii $j+1$ o jedna. Teda hodnoty pre každú podpostupnosť vieme vyrátať v konštantnom čase – stačí nám jediná operácia. Podpostupností je spolu zhruba N^2 a takáto je aj výsledná časová zložitosť tohto algoritmu – $O(N^2)$.

Všimnime si ešte jednu vec. Ak už podpostupnosť od i po j obsahuje všetky tri písmená `a, b, c`, tak aj všetky dlhšie podpostupnosti začínajúce na pozícii i budú obsahovať všetky 3 písmená. Nasleduje kód, ktorý robí to, čo sme si práve popísali:

```
vysl := N + 1;
for i := 1 to N do
begin
    for j := 0 to 2 do p[j] := 0;
    for j := i to N do
    begin
        inc(p[ord(a[j]) - ord('a')]);
        if (p[0] > 0) and (p[1] > 0) and (p[2] > 0) then
        begin
            if (j - i + 1 < vysl) then vysl := j - i + 1;
            break;
        end;
    end;
end;
if (vysl <= N) then writeln(vysl)
else writeln('Neda sa.');
```

V programe kontrolujeme, či už postupnosť od i do j obsahuje všetky 3 písmená. Ak áno, tak vyskočíme z cyklu pomocou príkazu `break`. Týmto sme zložitosť algoritmu nezlepšili, ale táto myšlienka vedie k rýchlejšiemu riešeniu. Všimnime si, kedy pre dané i vyskakujeme z cyklu. Platí, že ak pre i vyskočíme z cyklu v momente, keď $j = t$, tak pre $i+1$ vyskočíme z cyklu najskôr v čase t.

Inými slovami, najkratšia dobrá postupnosť začínajúca $(i+1)$, znakom nemôže končiť skôr ako najkratšia dobrá postupnosť začínajúca i-tym znakom.

Označme $f(i)$ také j, pre ktoré vyskočíme z cyklu. Ak z cyklu nikdy nevyskočíme, tak $f(i) = N + 1$. Náš vzorový program bude fungovať tak, že bude postupne rátať hodnoty $f(i)$. Ako sme si už ukázali vyššie, bude platiť nerovnosť $1 \leq f(1) \leq f(2) \leq \ldots \leq f(N) \leq N + 1$. Vďaka tejto nerovnosti vieme všetky tieto hodnoty vyrátať v lineárnom čase. Hodnotu $f(i+1)$ rátame tak, že začneme s hodnotou $f(i)$ a posúvame sa doprava, kým nevyhovuje. Doprava sa môžeme posunúť najviac o N, takže spolu vykonáme $O(N)$ operácií.

Ešte raz poriadnejšie popíšeme, čo sa stane na konci jednej iterácie cyklu pre `i`. Práve sme zistili hodnotu $f(i)$, vieme teda, že najkratší dobrý úsek, ktorý začína na pozícii i, končí na pozícii $f(i)$. A nielen to, my aj vieme presné počty písmen a, b a c v tomto úseku. Keď sa teraz pozrieme na znak `a[i]`, vieme z neho určiť presné počty a, b a c v úseku od $i+1$ po $f(i)$. Ak sú všetky tieto počty kladné, je $f(i+1) = f(i)$ a ide ďalšia iterácia vonkajšieho cyklu. Ak nie, ďalej zvyšujeme index konca postupnosti, až kým nebudú všetky tri počty znova kladné (alebo sa neminie vstup).

Listing programu:

```
var p : array[0..2] of integer;
    vysl, N, i, f : integer;
    x : string;
begin
    readln(N);
    readln(x);
    vysl := N + 1;
    f := 1;
    for i := 0 to 2 do p[i] := 0;
    for i := 1 to N do begin
        while f <= N do begin
            inc(p[ord(x[f]) - ord('a')]);
            if (p[0] > 0) and (p[1] > 0) and (p[2] > 0) then begin
                if (f - i + 1 < vysl) then vysl := f - i + 1;
                break;
            end;
            inc(f);
        end;
        dec(p[ord(x[i]) - ord('a')]);
    end;
    if (vysl <= N) then writeln(vysl)
    else writeln('Neda sa.');
end.
```

Alternatívne riešenie:

O niečo pomalšie, ale taktiež efektívne riešenie je pre každé $x \in \{a, b, c\}$ a každé $i \in \{0, \ldots, N\}$ spočítať $p_{x,i}$: počet výskytov znaku x v prvých i znakoch vstupu. Tieto hodnoty vieme jednoducho spočítať v čase $O(N)$. A pomocou týchto hodnôt vieme k ľubovoľnému začiatku úseku i efektívne nájsť najbližší koniec j, pre ktorý už bude úsek od i po j dobrý. Hľadáme totiž najmenšie j také, že pre všetky tri x je $p_{x,j} > p_{x,i-1}$. (V úseku $1 \ldots j$ je viac výskytov znaku x ako v úseku $1 \ldots i-1$ práve vtedy, keď je v úseku $i \ldots j$ aspoň jedno x.) A najmenšie takéto j vieme nájsť v čase $O(\log N)$ binárnym vyhľadávaním v intervale od i po N. Celková časová zložitosť takéhoto riešenia je teda $O(N \log N)$.

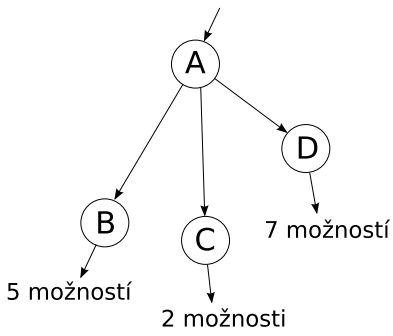
B-II-3 Sánkovanie

Riešenie „na papieri“:

Na úvod sa zamyslime, ako by sa túto úlohu dalo šikovne riešiť, keby sme ju mali riešiť ručne. Niekto nám dá na papieri obrázok toho, ako to na kopci vyzerá, a my máme spočítať všetky cesty zhora dole.

Ide to samozrejme aj skúšaním všetkých možností, ľahko však prídeme na lepšiu myšlienku:

Všimnime si situáciu z obrázka vpravo. Keď sme práve na mieste A, máme na výber tri možnosti: ísť do B, do C, alebo do D. Ak by sme už vedeli, koľkými spôsobmi sa dá dostať do cieľa z každého z týchto miest, tak už vieme všetko, čo potrebujeme. Počet ciest z A do cieľa je jednoducho rovný súčtu počtov ciest z B, C a D do cieľa. V príklade na obrázku vpravo by sa teda z A dalo do cieľa dostať $5 + 2 + 7 = 14$ spôsobmi.



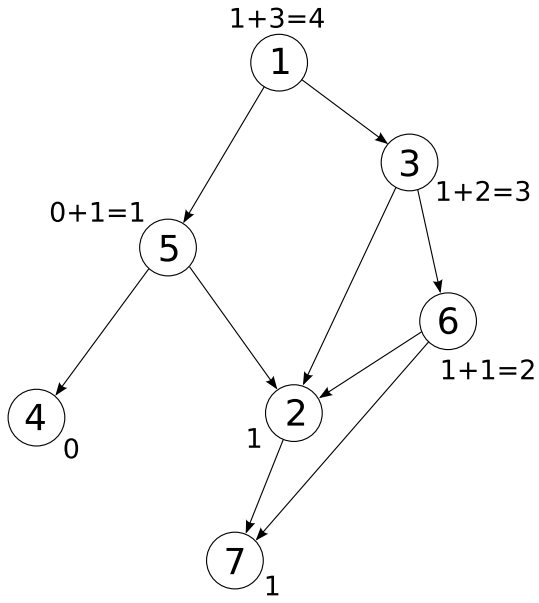
(Uvedomte si, že na ničom inom ako možnostiach, ktoré máme v prvom kroku, nám pri počítaní ciest z A nezáleží. Je nám napríklad úplne jedno, či bola možnosť ísť z D do C – ak aj bola, je už zarátaná v počte všetkých ciest z D do cieľa.)

Ručne teda odpoveď spočítame ľahko – stačí ísť po obrázku zdola hore a postupne pre každé miesto spočítať počet ciest, ktoré z neho vedú do cieľa.

Počty ciest, ktoré by sme dostali pre kopec z príkladu v zadaní, sú spolu s postupom ich počítania znázornené na druhom obrázku.

Prepísanie myšlienky do algoritmu:

Jeden spôsob, ako našu úlohu vyriešiť, je rozdeliť si problém na dve časti. Najskôr si usporiadame miesta zdola hore, a následne spočítame počty ciest.



V tomto okamihu si treba uvedomiť, že na základe vstupných údajov sa nemusí dať zoradiť miesta podľa výšky jednoznačne. Napríklad pre príklad zo zadania vyhovuje aj poradie $7, 2, 6, 3, 4, 5, 1$, aj poradie $7, 4, 2, 5, 6, 3, 1$, aj veľa iných.

Dobrá správa však je, že nám je úplne jedno, ktoré poradie si vyberieme, všetky sú pre nás dobré. Jediné, čo potrebujeme, je, aby každé miesto bolo v nájdenom poradí skôr ako všetky, z ktorých sa doň vieme dostať.

Terminologická vsuvka. Matematickému modelu kopca zo zadania sa hovorí orientovaný acyklický graf. *Zaujímavé miesta sa volajú* vrcholy *a čiary medzi nimi sú* hrany. *Slovo* orientovaný *znamená, že každá hrana má smer a slovo* acyklický, *že tam nie sú cykly – okruhy, po ktorých by sa dalo sánkovať dookola. To, čo chceme nájsť, sa odborne volá* topologické usporiadanie *vrcholov.*

Jedna možnosť, ako zostrojiť topologické usporiadanie vrcholov, je nasledovná. Pre každý vrchol si budeme pamätať dve veci: Počet hrán, ktoré doň vchádzajú, a čísla vrcholov, kam z neho vedú hrany.

Teraz si stačí uvedomiť, že akonáhle do niektorého vrcholu nevchádza žiadna hrana, môžeme ho prehlásiť za najvyšší, nič tým nepokazíme. Teraz zoberieme vrchol, ktorý sme práve spracovali, odstránime ho, a odstránime aj všetky hrany, ktoré z neho vedú. Čo sa stane? Môžu nám vzniknúť iné vrcholy, do ktorých nič nepovedie. Napríklad v grafe z príkladu v zadaní po tom, ako prehlásime 1 za najvyšší vrchol, už budú takéto vrcholy dva: 3 a 5.

A takto pokračujeme dokola ďalej, až kým sa nám všetky vrcholy neminú. V príklade by sme teraz napríklad mohli za druhý najvyšší prehlásiť vrchol 3, čím by nám pribudol nový kandidát: vrchol 6.

(Ak by mohol byť v grafe cyklus, časom by sme sa zasekli na tom, že do každého vrcholu už vchádza nejaká hrana. Ale keďže máme zaručené, že náš graf cykly neobsahuje, toto nepotrebujeme nijako ošetrovať.)

Dobrý spôsob, ako tento algoritmus implementovať, je pamätať si v ľubovoľnej rozumnej dátovej štruktúre (napríklad fronta alebo zásobník) množinu všetkých kandidátov. Aby bol náš algoritmus efektívny, stačí nám vedieť robiť v konštantnom čase dve operácie – pridať do množiny nového kandidáta a vybrať z množiny jedného (ľubovoľného) kandidáta.

Reprezentácia grafu v počítači:

Najjednoduchší spôsob, ako reprezentovať graf, je pole rozmerov $N \times N$, kde je na súradniciach i, j zapísané, či existuje hrana z i do j. Takýto prístup je dobrý vtedy, ak je graf na vstupe „hustý“.

Lenže všimnime si, čo by sa stalo pre $N = 100\,000$. Aj ak by sme použili len jeden bit pamäte na hranu, ešte stále by sme potrebovali $100\,000^2/8$ bajtov $\sim 1.2TB$.

Navyše, ak by sme aj mali maximálny počet hrán $M = 1\,000\,000$, tak by v priemere z jedného vrcholu išlo 10 hrán. Ale my by sme museli prejsť 100 000 políčok v tabuľke na to, aby sme tých 10 hrán našli.

Šikovnejšia reprezentácia grafu je pamätať si presne to, čo využívame pri topologickom triedení – *výstupné stupne vrcholov* (t. j. počty hrán, ktoré z nich vedú) a pre každý vrchol zoznam vrcholov, kam z neho vedú hrany.

Implementácia riešenia používajúceho topologické triedenie:

Ukážeme si teraz, ako všetko, čo sme doteraz vymysleli, implementovať v Pascale. V našom programe najskôr všetky hrany načítame do poľa, spočítame si stupne vrcholov, potom príslušne pozväčšujeme a vyplníme potrebné polia. Následne vyrobíme vyššie popísaným postupom jedno topologické usporiadanie vrcholov, a na záver ich v tomto poradí spracujeme a vypočítame hľadaný počet ciest.

Listing programu:

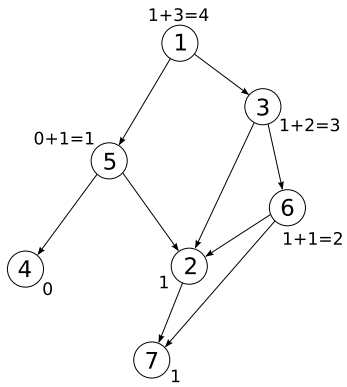
```
const MAXN = 100047; MAXM = 1000047;

var D : array[1..MAXN] of longint; { stupne vrcholov }
    G : array[1..MAXN] of array of longint; { G[i] je zoznam potomkov vrcholu i }
    T : array[1..MAXN] of longint; { topologicke usporiadanie vrcholov }
    N,M : longint;
    E : array[1..MAXM,1..2] of longint; { pomocne pole na hrany }
    Dpom : array[1..MAXN] of longint; { pomocne pole na stupne vrcholov }
    K : array[1..MAXN] of longint; { kandidati na dalsi najvyssi vrchol }
    P : array[1..MAXN] of longint; { P[i] je pocet ciest z i do N }

procedure nacitaj;
var i : longint;
```

```
begin
   { nacitame vstup }
   readln(N); readln(M);
   for i:=1 to M do readln(E[i][1],E[i][2]);
   { spocitame stupne vrcholov }
   for i:=1 to N do begin D[i]:=0; Dpom[i]:=0; end;
   for i:=1 to M do inc( D[ E[i][1] ] );
   { zvacsime riadky pola G na spravnu velkost }
   for i:=1 to N do setlength(G[i],D[i]);
   { vyplnime pole G }
   for i:=1 to M do begin
      G[ E[i][1] ][ Dpom[ E[i][1] ] ] := E[i][2];
      inc( Dpom[ E[i][1] ] );
   end;
end;

procedure utried;
var i, j, kde, PK : longint; { PK je pocet kandidatov }

begin
   { inicializacia: vyplnime Dpom, pridame ako kandidatov vsetkych co uz mozu }
   for i:=1 to N do Dpom[i]:=0;
   for i:=1 to N do for j:=0 to D[i]-1 do inc( Dpom[G[i][j]] );
   PK:=0;
   for i:=1 to N do if Dpom[i]=0 then begin inc(PK); K[PK]:=i; end;
   { dokola vyberieme a spracujeme kandidata, az kym sa vsetci neminu }
   for j:=1 to N do begin
      kde:=K[PK]; dec(PK); T[N+1-j]:=kde;
      { "kde" je prave spracuvany kandidat, ideme odstranit hrany z neho }
      for i:=0 to D[kde]-1 do begin
         dec(Dpom[ G[kde][i] ]);
         { ak tento pocet klesol na nulu, mame noveho kandidata }
         if Dpom[ G[kde][i] ]=0 then begin inc(PK); K[PK]:=G[kde][i]; end;
      end;
   end;
end;

procedure spocitaj;
var i, j, kde : longint;
begin
   for i:=1 to N do begin
      kde := T[i];
      if kde=N then P[kde]:=1 else P[kde]:=0;
      for j:=0 to D[kde]-1 do inc(P[kde],P[ G[kde][j] ]);
   end;
end;

begin
   nacitaj; utried; spocitaj; writeln(P[1]);
end.
```

Jednoduchšie implementovateľné riešenie:

Na ten istý problém sa však dá pozrieť aj z opačnej strany. Na základe pozorovania z prvého obrázku vieme napísať jednoduchú rekurzívnu funkciu, ktorá bude počítať počet ciest. Vyzerala by nejak takto:

```
function pocet_ciest(odkial : longint) : longint;
begin
   if odkial=N then result:=1 { som dole }
   else begin
      result:=0;
      for i := 1 to stupen[odkial] do
         result := result + pocet_ciest( potomok[odkial][i] );
   end;
end;
```



Keby sme takúto funkciu napísali poriadne a spustili, zistili by sme, že síce počet ciest zráta správne, ale zato veľmi pomaly – totiž postupne vyskúša všetky možné cesty, a tých môže byť vééééľmi veľa.

No ale my máme v rukáve jeden jednoduchý, ale o to účinnejší trik, odborne zvaný *memoizácia*. Stačí si uvedomiť, že akonáhle raz zistíme počet ciest z nejakého vrcholu, môžeme si ho zapísať do poľa s výsledkami. A akonáhle budeme túto hodnotu niekedy v budúcnosti potrebovať, namiesto opätovného skúšania všetkých možností sa pozrieme do poľa, zistíme, že už vieme odpoveď, a rovno ju vrátime.

Takto by celý beh algoritmu vyzeral na príklade zo zadania.

- zavoláme `pocet_ciest(1)`
- zavoláme `pocet_ciest(5)`
- zavoláme `pocet_ciest(4)`
- `pocet_ciest(4)` spočíta, zapamätá si a vracia 0
- zavoláme `pocet_ciest(2)`
- zavoláme `pocet_ciest(7)`
- `pocet_ciest(7)` spočíta, zapamätá si a vracia 1
- `pocet_ciest(2)` spočíta, zapamätá si a vracia 1
- `pocet_ciest(5)` spočíta, zapamätá si a vracia 1
- zavoláme `pocet_ciest(3)`
- zavoláme `pocet_ciest(2)`

- `pocet_ciest(2)` **rovno vracia zapamätanú hodnotu 1**
- zavoláme `pocet_ciest(6)`
- zavoláme `pocet_ciest(2)`
- `pocet_ciest(2)` **rovno vracia zapamätanú hodnotu 1**
- zavoláme `pocet_ciest(7)`
- `pocet_ciest(7)` **rovno vracia zapamätanú hodnotu 1**
- `pocet_ciest(6)` spočíta, zapamätá si a vracia 2
- `pocet_ciest(3)` spočíta, zapamätá si a vracia 3
- `pocet_ciest(1)` spočíta, zapamätá si a vracia 4

A akoby zázrakom dostávame z pôvodne mizerného algoritmu efektívny. Všimnite si totiž, že každý vrchol spracujeme a vypočítame práve raz. Tiež si všimnite poradie, v akom náš algoritmus zistil a uložil vypočítané hodnoty: ako prvý sme sa dozvedeli vrchol 4, potom 7, potom 2, 5, 6, 3, a nakoniec 1. Toto samozrejme nie je nič iné ako jedno možné topologické usporiadanie, ktoré sme dostali úplne zadarmo ako bonus k riešeniu zadanej úlohy.

Listing programu:

```
const MAXN = 100047; MAXM = 1000047;

var D : array[1..MAXN] of longint; { stupne vrcholov }
    G : array[1..MAXN] of array of longint; { G[i] je zoznam potomkov vrcholu i }
    N,M : longint;
    E : array[1..MAXM,1..2] of longint; { pomocne pole na hrany }
    Dpom : array[1..MAXN] of longint; { pomocne pole na stupne vrcholov }
    P : array[1..MAXN] of longint; { P[i] je pocet ciest z i do N }

procedure nacitaj;
var i : longint;
begin
   readln(N); readln(M);
   for i:=1 to M do readln(E[i][1],E[i][2]);
   for i:=1 to N do begin D[i]:=0; Dpom[i]:=0; end;
   for i:=1 to M do inc( D[ E[i][1] ] );
   for i:=1 to N do setlength(G[i],D[i]);
   for i:=1 to M do begin
      G[ E[i][1] ][ Dpom[ E[i][1] ] ] := E[i][2];
      inc( Dpom[ E[i][1] ] );
   end;
   for i:=1 to N do P[i]:=-1; { DOLEZITE! inicializujeme pole P na "neviem" }
end;

function pocet_ciest(kde : longint) : longint;
var vysledok, i, j : longint;
```

```
begin
   if P[kde] >= 0 then vysledok := P[kde] else
   if kde = N then vysledok := 1 else begin
      vysledok := 0;
      for i:=0 to D[kde]-1 do inc( vysledok, pocet_ciest(G[kde][i]) );
   end;
   P[kde] := vysledok; pocet_ciest := vysledok;
end;

begin
   nacitaj;
   writeln(pocet_ciest(1));
end.
```

Časová zložitosť riešení a poznámky k implementácii:

Obe uvedené implementácie majú časovú aj pamäťovú zložitosť $O(M+N)$.

Pamäťová zložitosť je zjavná, naozaj si vystačíme s konečným počtom M- a N-prvkových polí, a ešte používame jedno dvojrozmerné pole `G`, ktoré dokopy obsahuje práve M záznamov.

Čo sa týka časovej zložitosti, tam si stačí uvedomiť, že pre každý vrchol len konečne veľa krát (dvakrát v prvom riešení, raz v druhom) spracujeme hrany, ktoré z neho vychádzajú.

Implementácia riešenia, ktoré sme prezentovali ako druhé v poradí, je výrazne stručnejšia ako u prvého riešenia. Pridáme ale pár varovných slov: Existuje veľa úloh, ktoré sa, podobne ako táto, dajú riešiť z oboch koncov – aj „zdola hore“ (takýto postup sa často volá dynamické programovanie), aj „zhora dole“ (to je práve predvedená memoizácia). Niekedy je lepší jeden prístup, niekedy druhý. Memoizácia sa väčšinou ľahšie implementuje. Takéto riešenie ale môže byť aj niekoľkokrát pomalšie od riešenia opačným smerom – máme totiž navyše réžiu s veľanásobným volaním funkcie.

A s tým súvisí aj druhé riziko – na vnorené volania funkcie treba mať dosť veľký zásobník (stack). Pri súčasných obmedzeniach v praxi (štandardná veľkost stacku je 8 MB) je 100 000 vnorení rekurzívnej funkcie približne na hranici toho, čo sa nám ešte na stack zmestí. Keby teda bola táto úloha zadaná ako praktická, nášmu druhému riešeniu by hrozilo, že na zákerných veľkých vstupoch môže namiesto správneho výsledku skončiť s chybou Stack overflow (Pretečenie zásobníka).

B-II-4 Banka

Podúloha a):

Stačí vziať čísla účtov 00, 11, 22, ..., 99. Ak sa v jednej cifre pomýlime, dostaneme číslo s dvoma rôznymi ciframi – teda nikdy takto nedostaneme iné číslo účtu.

Podúloha b):

Všetkých možných N-ciferných čísel účtu je 10^N.

Rozdeľme ich do 10^{N-1} skupín podľa prvých $N-1$ cifier. V každej skupine teda dostaneme 10 čísel, ktoré sa navzájom líšia len v poslednej cifre.

Teraz si už len stačí uvedomiť, že z každej skupiny môžeme použiť nanajvýš jedno číslo – ak by sme z niektorej použili dve, zjavne by sa mohlo stať, že sa pri písaní jedného z nich pomýlime v poslednej cifre a napíšeme to druhé.

Podúloha c):

Najskôr ukážeme ľahšie vymysliteľný postup, ktorým dostaneme $\lceil 10^N/11 \rceil$ vyhovujúcich čísel účtu. Tento postup bude zovšeobecnením riešenia, ktoré sme našli v prvej podúlohe: Stačí ako čísla účtov brať násobky čísla 11.

Prečo to funguje? Z podobného dôvodu ako funguje kritérium deliteľnosti 11, s ktorým ste sa už možno stretli: číslo je deliteľné 11 práve vtedy, ak je 11 deliteľný jeho ciferný súčet so striedavými znamienkami. Obe veci si teraz dokážeme.

Zapíšme naše číslo účtu ako $U = \overline{a_{n-1}a_{n-2}\ldots a_1 a_0}$. Hodnota tohto čísla je $a_0 + 10a_1 + 10^2 a_2 + \cdots + 10^{n-1}a_{n-1}$. Teraz si stačí uvedomiť, že 10 dáva po delení 11 rovnaký zvyšok ako -1. A teda pre ľubovoľné k dáva 10^k rovnaký zvyšok ako $(-1)^k$. A samozrejme vieme, že pre nepárne k to je -1 a pre párne to je 1.

Tým sme práve dokázali, že naše číslo účtu dáva rovnaký zvyšok po delení 11 ako $U' = a_0 - a_1 + a_2 - \cdots + (-1)^{n-1}a_{n-1}$. Z tohto priamo vyplýva vyššie spomínané kritérium deliteľnosti 11.

A ako je to teda s číslami účtov? Všimnime si, že ak zmeníme v čísle U ľubovoľnú jednu cifru, hodnota U' sa určite zmení. A keďže cifru vieme zmeniť najviac o 9 (z 0 na 9 alebo naopak), zmena jednej cifry určite zmení zvyšok, aký dáva číslo po delení 11. To ale znamená, že ak sme pred zmenou mali číslo, ktoré bolo deliteľné 11, po zmene cifry už 11 deliteľné nebude. A to je presne to, čo potrebujeme.

Práve popísaná konštrukcia je síce dobrá, ale nie je optimálna. Optimálne

riešenie, ktoré si popíšeme, bude založené na myšlienke *kontrolného súčtu*.

Čísla účtov budeme voliť takto: Prvých $N-1$ cifier zvolíme ľubovoľne, a následne poslednú zvolíme tak, aby bol ciferný súčet nášho čísla násobkom 10.

Poslednú cifru vždy vieme určiť jednoznačne – stačí sčítať prvých $N-1$ cifier a pozrieť sa na poslednú cifru výsledku, označme ju x. Ak $x = 0$, je posledná cifra čísla účtu 0, inak je to $10 - x$.

Týmto spôsobom teda dostaneme 10^{N-1} čísel účtov. A zjavne platí, že ak v čísle účtu zmeníme jednu cifru, zmeníme tým jeho ciferný súčet o 1 až 9. A teda ak pôvodné číslo malo ciferný súčet deliteľný 10, nové číslo určite nebude mať ciferný súčet deliteľný 10.

Podúloha d):

Obe banky majú presne rovnaký maximálny počet účtov, a to nie len pre 10-ciferné čísla účtov, ale dokonca pre ľubovoľné N. Požiadavky zo zadania sú totiž ekvivalentné. Ukážeme, prečo.

Požiadavka z Popletenej Viesky je zjavne splnená práve vtedy, keď sa každé dve čísla účtov líšia aspoň v piatich cifrách. (Ak by existovala dvojica čísel, ktorá sa líší na menej miestach, vieme max. 4 chybami z jedného vyrobiť druhé. V opačnom prípade sa to zjavne stať nemôže.)

Lenže aj požiadavka z Bezpečnej Triesky je splnená práve vtedy, keď sa každé dve čísla účtov líšia aspoň v piatich cifrách.

Totiž množina čísel účtov je pre túto banku zlá, ak existujú nejaké dva účty U_1 a U_2 také, že nejaké číslo U sa dá na nejaké ≤ 2 zmeny vyrobiť z U_1 a na nejaké iné ≤ 2 zmeny vyrobiť z U_2.

Ak také U existuje, tak vieme na ≤ 2 zmeny prerobiť U_1 na U, a na ďalšie ≤ 2 zmeny prerobiť U na U_2, preto sa U_1 a U_2 líšia na ≤ 4 miestach. A naopak, ak sa U_1 a U_2 líšia na $x \leq 4$ miestach, také U existuje – stačí zobrať U_1 a ľubovoľných $\lceil x/2 \rceil$ cifier kde sa líši od U_2 zmeniť na príslušné cifry U_2.

Tým sme ukázali, že množina čísel účtov je pre túto banku zlá práve vtedy, keď sa niektoré dve čísla účtov líšia na ≤ 4 miestach. Inými slovami, množina čísel účtov je dobrá práve vtedy, keď sa každé dve čísla účtov líšia aspoň v piatich cifrách.

(*Táto podúloha ukazuje dôležité pozorovanie z teórie samoopravných kódov: Ľubovoľný kód, ktorý dokáže rozpoznať $2k$ chýb, je zároveň kódom, ktorý dokáže opraviť k chýb, a naopak.*)

Riešenia celoštátneho kola kategórie A

A-III-1 Horár Jedlička II

Úlohu vyriešime metódou *zametania*, ktorá sa používa často pre geometrické problémy: Predstavíme si priamku p rovnobežnú s osou x, ktorá sa pohybuje v smere osi y (od záporných čísel ku kladným – tomuto smeru budeme hovoriť zdola nahor). Budeme udržiavať prienik tejto priamky so zadanými úsečkami. Pre tento účel budeme používať nasledovné dátové štruktúry:

- Zoznam úsečiek S pretínajúcich v danom okamihu priamku p, v poradí podľa x-ovej súradnice týchto priesečníkov. V tomto zozname budeme potrebovať rýchlo vyhľadávať, preto ho budeme realizovať ako (vyvažovaný) vyhľadávací strom. Za zmienku stojí, že si v tejto dátovej štruktúre nepamätáme súradnice priesečníkov s priamkou p. Tieto súradnice sa menia, keď sa priamka p posunie, a ich upravovanie by bolo príliš pomalé. Namiesto toho si pamätáme iba poradie týchto priesečníkov. Toto poradie sa mení iba keď p prejde jedným z priesečníkov úsečiek.

  Naviac vieme, že keď takáto situácia nastane, iba sa obráti poradie priesečníkov priamky p s úsečkami, ktoré sa v danom bode pretínali. Súradnice priesečníkov s p dopočítavame až v priebehu vyhľadávania z aktuálnej y-ovej súradnice p.

  Tiež si uvedomte, že pre ľubovoľný bod priamky p vieme v čase $O(\log N)$ zistiť, medzi ktorými dvoma úsečkami sa nachádza.

- Fronta udalostí U, ktoré ovplyvňujú prienik p s úsečkami:

  – Spodné konce úsečiek, ktoré sú celé nad p.
  – Priesečník každej dvojice úsečiek susediacich v S, ak existuje a leží nad p.
  – Vodorovné úsečky. Tie treba ošetriť špeciálne.
  – Horné konce úsečiek, ktoré sú nad p.

  Frontu udalostí môžeme realizovať napr. ako haldu. (Prípadne môžeme opäť použiť vyvažovaný binárny strom.) Udalosti porovnávame podľa ich y-ovej súradnice a v prípade zhody podľa poradia v predošlom zozname.

Na začiatku je priamka p v $-\infty$, teda zoznam S je prázdny a fronta U obsahuje horné a dolné konce všetkých úsečiek. Pre zjednodušenie niektorých operácií pridajme do S dve falošné úsečky rovnobežné s osou y, v $-\infty$ a $+\infty$ (alebo v ľubovoľných bodoch ležiacich naľavo a napravo od všetkých ostatných úsečiek). V priebehu simulácie odoberáme z U najmenšiu udalosť a upravujeme S a U podľa nej. Program sa končí vo chvíli, keď priamka p dorazí do $+\infty$, teda keď je U prázdna.

Vždy, keď do S pridáme prvok, pridáme tiež do U udalosť prieniku so susednými úsečkami, pokiaľ prienik existuje a leží nad p.

Popíšeme teraz detailnejšie, ako spracovávame udalosť spôsobenú úsečkou u. Udalosti s rovnakou y-ovou súradnicou spracovávame v tomto poradí:

- *Spodný koniec úsečky u:* Pridáme u do S tak, aby prieniky jednotlivých úsečiek boli stále usporiadané.
- *Vodorovná úsečka $u = (x_1, y) - (x_2, y)$:* Vypíšeme priesečníky s úsečkami z S, ktorých x-ová súradnica na priamke p je z intervalu $\langle x_1, x_2 \rangle$.
- *Priesečník dvoch úsečiek v bode x, y:* Zmažeme z S všetky úsečky prechádzajúce bodom (x, y) a dáme ich nabok. Vypíšeme (x, y).
- S je teraz utriedené podľa priesečníkov s priamkou p (môže sa zmeniť iba poradie tých úsečiek, ktoré sa pretínajú na p, ale tie sme dali nabok). Do S teraz zatriedime úsečky, ktoré sme dali nabok.
- *Horný koniec úsečky u:* Nech u leží medzi úsečkami u_1 a u_2. Odstránime u z S. Naviac, pokiaľ sa u_1 a u_2 pretínajú nad p, pridáme ich priesečník do U.

Treba ešte doriešiť tento technický detail: *zaokrúhľovacie chyby* – pokiaľ by spôsobili zámenu poradia dvoch udalostí v U alebo dvoch úsečiek v S, mohlo by to viesť k zlému výsledku. Preto je nutné všetky porovnania robiť presne. To sa dá dosiahnuť viacerými spôsobmi, jednou z možností je počítať súradnice priesečníkov ako zlomky.

Pamäťová zložitosť je lineárna – ako U tak S vždy obsahujú $O(N)$ prvkov. Aká je časová zložitosť? Nech P je počet priesečníkov úsečiek. Každá operácia s U či S zaberie čas $O(\log N)$. Spočítajme teraz počet týchto operácií pre každú z udalostí:

- *Spodný alebo horný koniec úsečky:* Týchto udalostí je $O(N)$ a vyžadujú konštantný počet operácií s U a S. Každá z nich spôsobuje pridanie najvýš dvoch udalostí do U.

- *Priesečník, v ktorom sa pretína k úsečiek:* Týchto udalostí je P a vyžadujú $O(k)$ operácií s S. Každý priesečník spôsobí pridanie nanajvýš dvoch a odobratie aspoň $k-1$ udalostí z U.

Počet udalostí pridaných do U (vrátane inicializácie) je $O(N+P)$ a počet odobraných udalostí je teda tiež $O(N+P)$. Počet operácií s S a U je obmedzený počtom udalostí pridaných či odobraných z U a je $O(N+P)$. Časová zložitosť je preto $O((N+P)\log N)$. Na záver uveďme, že existuje aj rýchlejšie (a podstatne zložitejšie) riešenie s časovou zložitosťou $O(N\log N+P)$ (Chazelle a Edeslbrunner, 1992).

Listing programu:

```
#include <stdio.h>
#include <stdlib.h>
#include <set>
#include <queue>
#include <algorithm>
using namespace std;

// Nejvetsi spolecny delitel dvou cisel
long long gcd(long long a, long long b) { return (b==0) ? a : gcd(b, a%b); }

// Zlomky a prace s nimi
struct fraction {
    long long nom, den;                          // citatel a jmenovatel
    fraction () { nom=0; den=1; }                // default je nula
    fraction(long long n) { nom=n; den=1; }      // cele cislo -> zlomek
    fraction(long long n, long long d) {         // uprava na zakladni tvar
        long long g = gcd(llabs(den), llabs(nom));
        nom=n/g; den=d/g;
        if (den<0) { nom=-nom; den=-den; }
    }
    fraction& operator =(long long b)  { nom=b; den=1; return *this; }
    fraction operator +(fraction b) const
      { return fraction(nom*b.den+b.nom*den, den*b.den); }
    fraction operator -(fraction b) const
      { return fraction(nom*b.den-b.nom*den, den*b.den); }
    fraction operator *(fraction b) const
      { return fraction(nom*b.nom, den*b.den); }
    fraction operator /(fraction b) const
      { return fraction(nom*b.den, den*b.nom); }
    bool operator  <(fraction b) const { return (*this-b).nom < 0; }
    bool operator ==(fraction b) const { return nom==b.nom && den==b.den; }
    bool operator !=(fraction b) const { return !(*this == b); }
    bool operator <=(fraction b) const { return (*this-b).nom <= 0; }
    double f() const { return (nom+0.0)/den; }     // konverze na realne cislo
};
```

```
fraction p(-900000);                                // zametaci primka

// Usecka ze vstupu
struct usecka {
    fraction x1, y1, x2, y2;                        // souradnice koncu
    bool prunik(usecka pr) const {                  // ma usecka prunik s jinou?
        fraction r = intersect(pr);
        return !(r<pr.y1 || pr.y2<r || r<y1 || y2<r);
    }
    fraction intersect(usecka pr) const {           // y-ova souradnice pruniku
        return (pr.dir() == dir()) ? 1000000 :
               (x(fraction(0)) - pr.x(fraction(0))) / (pr.dir() - dir());
    }
    fraction x(fraction y) const {                  // x-ova souradnice pro y=p
      return x1+dir()*(y-y1);
    }
    fraction dir() const { return (x2-x1)/(y2-y1); } // smernice
    bool operator < (const usecka &pr) const { // porovnani dle pruseciku s p
        if (x(p) != pr.x(p)) return x(p) < pr.x(p); else return dir() < pr.dir();
    }
};

// Udalost
struct event{
    fraction y;                                     // kdy nastava
    int type;   // typ: 0 zacatek, 1 krizeni, 2 vodorovna primka, 3 konec
    usecka u;
    bool operator < (const event &e) const {
        if (e.y != y) return e.y < y; else return type > e.type;
    }
};

priority_queue <event> U;           // fronta udalosti
set <usecka> S;                     // zadane usecky
typedef set <usecka>::iterator iter; // iterator pres usecky
int n;                              // kolik je usecek
usecka prk[1000000];                // pomocne pole na zmenene primky

// Pokud se a s b pronika nad zametaci primkou, vytvorime udalost.
void pronika(usecka a, usecka b) {
    if (!a.prunik(b)) return;
    event f; f.y = a.intersect(b); f.u = b; f.type = 1;
    if (p < f.y) U.push(f);
}

void pridej(usecka u) {
    iter it = S.insert(u).first;    // iterator na pridany prvek
    it--; pronika(*it, u);
    it++; it++; pronika(u, *it);
```

```
}

int main() {
    int x1, y1, x2, y2, i;
    scanf("%i", &n);
    for (i=0; i<n; i++){
        usecka u;
        scanf("%i%i%i%i" , &x1, &y1, &x2, &y2);
        if (y2 < y1) { swap(x1, x2); swap(y1, y2); }
        if (y1 == y2) {
            if (x2 < x1) swap(x1, x2);
            u.x1 = x1; u.y1 = y1; u.x2 = x2; u.y2 = y2;
            event e; e.u = u; e.type = 2; e.y = u.y1;
            U.push(e);
        } else {
            event e;
            u.x1 = x1; u.y1 = y1; u.x2 = x2; u.y2 = y2;
            e.u = u;
            e.type = 0; e.y = u.y1; U.push(e);
            e.type = 3; e.y = u.y2; U.push(e);
        }
    }

    event ev; ev.y = 1000000; U.push(ev);       // zarazky
    usecka u;
    u.y2 = u.x1 = u.x2 = 1000000; u.y1 = -1000000; S.insert(u);
    u.y1 = u.x1 = u.x2 = -1000000; u.y2 = 1000000; S.insert(u);

    while (1) {
        event e = U.top(), f;
        fraction y = e.y;                       // nova poloha zametaci primky
        if (y == fraction(1000000)) break;      // zarazka => konec
        while (y == e.y && e.type == 0) {       // nove primky
            pridej(e.u);
            U.pop(); e=U.top();                 // dalsi udalost
        }

        int k = 0;
        while (y == e.y && e.type == 1) {       // pruseciky
            iter it = S.find(e.u), it2;
            if (it != S.end()) {
                // smazeme vse, co se meni
                fraction x = it->x(y);
                printf("%f %f\n", x.f(), y.f());
                while (it->x(y) == x) it--;
                it++;
                while (it->x(y) == x) {
                    it2 = it; it2++;
                    prk[k++] = *it;             // a zapamatujeme si, co se meni
                    S.erase(it);
```

```
                it = it2;
            }
        }
        U.pop(); e = U.top();
    }

    while (p == e.y && e.type == 2) {       // vodorovne primky
        usecka u;
        u.y1 = -1000000; u.y2 = 1000000;
        u.x1 = u.x2 = e.u.x1;
        iter i = S.lower_bound(u);
        for (iter i = S.lower_bound(u); i->x(y) <= e.u.x2; i++)
            printf("%f %f\n", i->x(y).f(), y.f());
        U.pop(); e = U.top();
    }

    p=y;                                    // posuneme zametaci primku
    for (i=0; i<k; i++) pridej(prk[i]);     // vratime zmenene
    while (y == e.y && e.type == 3) {       // konce primek
        iter it = S.find(e.u), it2 = it;
        it++; it2--;
        pronika(*it, *it2);
        S.erase(e.u);
        U.pop(); e = U.top();
    }
  }
  return 0;
}
```

A-III-2 Magické cestovanie

Našou úlohou je nájsť najdlhšiu súvislú podpostupnosť zadanej postupnosti nezáporných celých čísel, ktorej súčet je násobok K. Inými slovami, hľadáme najdlhšiu súvislú podpostupnosť, ktorej súčet dáva po delení číslom K zvyšok 0. V celom tomto riešení budeme všetky matematické operácie robiť modulo K, teda ak napríklad $K = 7$, tak súčet postupnosti $2, 4, 2$ je 1.

Predstavme si najskôr, že hľadaná podpostupnosť môže začínať len na začiatku. Potom by sa nám úloha riešila jednoducho – postupne by sme spracúvali čísla v postupnosti a počítali si ich súčet. Zakaždým, keď by sme dostali súčet 0, našli sme potenciálne riešenie. Keďže chceme najdlhšiu takú podpostupnosť, výsledkom by bolo posledné takéto miesto.

Zostáva už len toto riešenie zovšeobecniť na podpostupnosť začínajúcu na ľubovoľnom mieste. Na to stačí spraviť nasledujúce pozorovanie. Nech sme práve spracovali j-te číslo v postupnosti a vieme, že súčet prvých j čísel je d. Ak pre

nejaké $i < j$ bol tiež súčet prvých i čísel rovný d, tak vieme, že podpostupnosť $a_{i+1}, \ldots, a_j$ musí mať súčet 0, a teda je možným riešením úlohy.

A to už je v podstate všetko, stačí si už len uvedomiť, že ak máme pre i viacero možností, chceme vybrať tú najmenšiu z nich – teda miesto, kde bol priebežne počítaný súčet čísel po prvý krát rovný d.

Aby sme takéto i vedeli efektívne nájsť, stačí si spraviť pole dĺžky K a v ňom si pre každý súčet $d, 0 \leq d < K$ zaznamenať, kde sa nachádzal prvý taký index i, pre ktorý sme dostali súčet d.

Na začiatku algoritmu musíme toto pole inicializovať – na všetky políčka si zapíšeme špeciálnu hodnotu (napr. -1), ktorá nám hovorí, že sa takýto súčet ešte nevyskytol. Následne stačí načítavať postupnosť, počítať priebežný súčet a zisťovať v poli, či sme už na daný súčet natrafili. Každý prvok postupnosti teda spracujeme v konštantnom čase. Celková časová zložitosť postupu je teda $O(N+K)$. Keďže si postupnosť nepotrebujeme pamätať, vystačíme si s pamäťou $O(K)$.

Malé cvičenie na záver: Vedeli by ste dokázať, že ak $N \geq K + 1$, tak určite existuje aspoň jedno riešenie?

Listing programu:

```
#include <cstdio>
using namespace std;
int N, K;             // hodnoty N a K ze vstupu
int zacatky[50007];   // nejmensi mozne indexy i s danym souctem s modulo K
int a, soucet = 0;    // pomocne promenne - nacteni vstupu a mezisoucet
int nej_zacatek = 0, nej_delka = 0;  // dosud nejlepsi nalezene reseni

int main() {
    scanf("%d%d", &N, &K);
    zacatky[0] = 1;
    for (int i = 1; i <= N; i++) {
        scanf("%d", &a);
        soucet = (soucet + a) % K;
        if (zacatky[soucet]) {
            if (i - zacatky[soucet] + 1 > nej_delka) {
                nej_zacatek = zacatky[soucet];
                nej_delka = i - nej_zacatek + 1;
            }
        } else zacatky[soucet] = i+1;
    }
    if (nej_delka) printf("%d %d\n", nej_zacatek, nej_zacatek + nej_delka - 1);
    else           printf("Nelze zaklinat.\n");
}
```

A-III-3 Zásobníkové počítače

Najjednoduhším riešením tejto úlohy bolo použiť 4 zásobníky, každý pre jeden znak, a na konci zistiť, či sú všetky zásobníky rovnako zaplnené. Po troške rozmýšľania sa dalo spraviť riešenie s troma zásobníkmi, pričom si v nich pamätáme len rozdiely počtov znakov: $a - b$, $a - c$, $a - d$. Všetky tieto riešenia sa dali naprogramovať s lineárnou časovou zložitosťou.

Použiť tri zásobníky je zbytočný luxus – totiž platí, že úplne čokoľvek, čo sa dá spraviť s použitím K zásobníkov, sa dá spraviť s použitím len 2 zásobníkov. Ukážeme teraz dve riešenia, ktoré potrebujú iba dva zásobníky.

Kvadratické riešenie:

Najskôr si načítame celý vstup do prvého zásobníka. Potom pomocou druhého zásobníka vymažeme z prvého zásobníka jeden znak a, jeden znak b, jeden znak c a jeden znak d. Toto budeme opakovať, až kým nebude prvý zásobník prázdny, alebo neminieme niektorý znak. Toto riešenie má zjavne kvadratickú časovú zložitosť.

Lineárne riešenie:

Ukážeme si teraz riešenie, ktoré bude používať dva zásobníky a bude mať lineárnu časovú zložitosť. Prvý zásobník bude fungovať ako štyri počítadlá a druhý zásobník bude pomocný. Každé počítadlo bude počítať počet výskytov jedného znaku. Najskôr si popíšeme, ako bude fungovať jedno počítadlo a potom si povieme, ako spojiť 4 počítadlá do jediného zásobníka.

Počítadlá budú binárne (číslo si budeme pamätať v dvojkovej sústave). Keď budeme chcieť počítadlo zvýšiť o jedna, tak použijeme klasický algoritmus na sčítanie čísliel po cifrách. Budeme prechádzať po čísliciach od najnižších bitov k najvyšším a prepisovať jednotky na nuly. Keď narazíme na prvú nulu, tak ju zmeníme na jednotku a skončíme (napríklad $110011 + 1 = 110100$). Ak by číslo skončilo skôr ako narazíme na nulu, tak na jeho začiatok dopíšeme ešte jednu jednotku ($111 + 1 = 1000$).

Ako tento postup naprogramovať pomocou zásobníkov? Počítadlo uložíme do zásobníka tak, že najpravejšia (najnižší bit) cifra bude na vrchole zásobníka. Môžeme postupne odoberať jednotlivé číslice sprava, prepisovať ich a odkladať ich do pomocného zásobníka. Keď budeme hotoví, tak presunieme obsah pomocného zásobníka na hlavný zásobník. Kód by vyzeral nasledovne:

Listing programu:

```
procedure zvys(var a:stack of 0..1); { zvys pocitadlo v zasobniku  a  o 1 }
var b: stack of 0..1;                { pomocny zasobnik }
```

```
begin
   repeat
      { ak treba, vlavo od cisla si domyslime nuly }
      if empty(a) then x := 0 else x := pop(a);
      push(b, (x+1) mod 2);                    { 0->1, 1->0 }
   until x=0;                                  { zastavime na prvej nule }
   while not empty(b) do push(a, pop(b));      { presypeme cisla spat }
end;
```

Ako uložiť 4 počítadlá do jedného zásobníka? Jedna možnosť je ukladať tam ich bity „na striedačku“. Druhá možnosť je použiť čísla z väčšieho rozsahu ako `0..1`. My ukážeme tú druhú z nich.

Do zásobníka budeme ukladať štvorbitové čísla, pričom i-te počítadlo bude zodpovedať i-temu najmenej významnému bitu každého z čísiel v zásobníku. Pri zvyšovaní počítadla budeme prepisovať len príslušné bity, ostatné bity (patriace iným počítadlám) necháme na pokoji.

Na prácu s bitmi sa nám budú hodiť operácie **and** a **xor**. Výraz a **and** b dáva prirodzené číslo, ktoré má v dvojkovom zápise na i-tom mieste 1 práve vtedy, ak aj a aj b majú na i-tom mieste 1. Podobne a **xor** b má jednotky tam, kde bola jednotka buď v a, alebo v b, ale nie v oboch naraz. Napríklad a **and** 8 je nula práve vtedy, ak na štvrtej pozícii čísla a je 0 a ináč je to 8. Číslo a **xor** 8 je číslo, v ktorom je oproti a zmenený štvrtý bit na opačný.

Celý program bude vyzerať nasledovne:

Listing programu:

```
program abcd;
var a, b: stack of 0..15; { zasobnik s pocitadlami, pomocny zasobnik }
    c: char;              { prave zpracovavany znak }
    m, x: 0..15;          { pomocne promenne }
begin
   while read(c) do begin
      if c='a' then m:=1        { ktory bit zodpoveda nacitanemu znaku? }
      else if c='b' then m:=2
      else if c='c' then m:=4
      else if c='d' then m:=8;
      repeat                    { zvysenie pocitadla o jedna }
         if empty(a) then x := 0 else x := pop(a);
         push(b, x xor m);
      until (x and m) = 0;
      while not empty(b) do push(a, pop(b)); { kopirujeme z "b" spat do "a" }
   end;
   m := 1;
   while not empty(a) do begin
      x := pop(a);
      if (x<>0) and (x<>15) then m := 0;
```

```
    end;
    write(m);
end.
```

Program používa len dva zásobníky a spotrebúva pamäť $O(\log N)$, lebo každé číslo menšie alebo rovné N má v dvojkovej sústave najviac $\lfloor \log_2 N \rfloor + 1$ číslic. Dĺžka čísla tiež obmedzuje počet opakovaní cyklu repeat, takže časová zložitosť nebude horšia ako $O(N \log N)$. Teraz si ukážeme, že časová zložitosť je v skutočnosti lineárna.

Rozbor stačí spraviť pre jedno počítadlo (teda pre našu ukážkovú procedúru `zvys`), keďže jednotlivé počítadlá sa navzájom neovplyvňujú a záverečné porovnanie trvá len $O(\log N)$. Naviac sa stačí zamerať len na cyklus repeat, lebo v cykle while trávime najviac toľko času ako v cykle repeat.

Uvažujme, čo sa stane, ak procedúru **zvys** zavoláme N krát po sebe, pričom na začiatku bolo počítadlo nastavené na 0. Sledujme, ako sa pri tom mení počet jednotiek v dvojkovom zápise počítadla. Operácie vo vnútri počítadla buď prepisujú 0 na 1 (pribudne jednotka), alebo 1 na 0 (jednotka ubudne). Naviac prepísaním 0 na 1 sa cyklus zastaví, takže po celú dobu výpočtu sa počet jednotiek zvýši najviac o N. Počet jednotiek ale nikdy neklesne na 0, takže operácii, ktoré ho znižujú nebude nikdy viac ako N. Preto je všetkých operácii $O(N)$.

Iná cesta ako dokázať lineárnosť nášho algoritmu: Keď si vypíšeme, koľko krokov spraví algoritmus počítajúci $N + 1$ pre $N = 0, 1, 2, \ldots$, dostaneme nasledujúcu postupnosť: $1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, \ldots$ Časová zložitosť nášho programu je priamo úmerná súčtu tejto postupnosti. Ten vieme odhadnúť nasledovne: každý člen je aspoň 1, každý druhý je aspoň 2, každý štvrtý je aspoň 3, atď., takže celkový súčet je rádovo rovný $N + N/2 + N/4 + \cdots = 2N$.

A-III-4 Výlet do Švajčiarska

Terminologický detail na úvod: Ak medzi dvoma mestami vedie možná etapa, budeme ich volať susedné.

Aby sme mohli uvádzať aj odhady časovej zložitosti riešení, označíme D maximálny počet susedov, ktoré môže mesto mať. Pripomíname, že v testovacích vstupoch pre počet miest platí $N \leq 10\,000$ a pre maximálny počet susedov $D \leq 100$.

Jednoduché riešenie:

Štvoríc miest predsa nemôže byť až tak veľa, nie? Čo tak ich všetky vyskúšať?

Spravíme si pole $N \times N$, kde si zaznačíme, ktoré dvojice miest sú spojené, a potom vyskúšame všetkých $N(N-1)(N-2)(N-3)$ štvoríc, pre každú overíme, či naozaj existujú všetky štyri potrebné etapy, a ak áno, spočítame ich celkové ohodnotenie.

Časová zložitosť tohto riešenia je $O(N^4)$ a pamäťová $O(N^2)$.

Takéto riešenie má hneď dva nedostatky. Prvý: počet štvoríc, ktoré treba vyskúšať, je približne $10\,000^4$. Pomocou jednoduchého pravidla „miliarda inštrukcií = sekunda“ môžeme odhadnúť, že pre maximálny vstup by tento program bežal tak okolo miliardy sekúnd. Alebo inými slovami: tento program by za sekundu zvládol zriešiť tak nanajvýš vstup s $N \leq 100$.

Druhým nedostatkom je spotreba pamäte. Aj keby sme tento program zázračne urýchlili, ešte stále nedostane plný počet bodov. Pre veľké N je totiž matica $N \times N$ priveľká, nezmestí sa do pamäťového limitu.

Trošku lepšie riešenie:

Problémov s pamäťou sa zbavíme, ak si nebudeme pamätať celú maticu $N \times N$, ale jednoducho si pre každé mesto zapamätáme jeho $\leq D$ susedov.

A už sama o sebe nám táto zmena pomôže dostať o niečo lepšie riešenie. Vyskúšame všetkých N možností pre prvé mesto, potom $\leq D$ možností pre druhé mesto (všetkých susedov prvého mesta), pre každú z nich máme $\leq D$ možností pre tretie mesto, a pre každú z nich $\leq D$ možností pre štvrté.

Časová zložitosť tohto riešenia je teda $O(ND^3)$ a pamäťová $O(ND)$.

Inými slovami, stačí nám vyskúšať $1\,000\,000N$ možností. Pre veľké N je to stále priveľa, ale už sme sa dostali z úrovne „maximálny vstup za miliardu sekúnd“ na úroveň „maximálny vstup za tisíc sekúnd“.

Ešte trochu sa dá ušetriť, keď si uvedomíme, že každý cyklus by sme pri tomto algoritme prezreli až osemkrát – raz pre každé mesto a každý smer. Jednoduchá optimalizácia je povedať si, že prvé mesto, ktoré skúšam, bude to, ktoré má na cykle najmenšie číslo. Takto už každý cyklus prezrieme len dvakrát, zrýchlime tým teda náš program približne na štvrtinu.

Vzorové riešenie:

Uvedomme si teraz, že cyklus nemusíme celý hľadať v smere, v ktorom po ňom bude Tibor prechádzať. Označme mestá na cykle, ktorý hľadáme, U, V, W a X. Na cyklus $U - V - W - X - U$ sa môžeme dívať ako na dve rôzne dvojetapové cesty $U - V - W$ a $U - X - W$.

Predstavme si, že sme už vybrali mesto U, kde chceme začínať a zároveň končiť. Pozrime sa na všetky dvojetapové úseky, ktoré v tomto meste začínajú.

Tých je nanajvýš D^2, lebo máme najviac D možností, kam ísť prvou etapou, a v jej cieli máme nanajvýš D možností, kam ísť druhou. O každom z týchto úsekov nás zaujímajú dve veci: mesto kde končí, a jeho ohodnotenie.

Chceme teraz vybrať mesto W a dva úseky z U do W tak, aby ich súčet ohodnotení bol čo najväčší. Aby sme toto vedeli spraviť, stačí si pre každé mesto, ktoré sa dá z U dosiahnuť dvoma etapami, pamätať (nanajvýš) dva najlepšie spôsoby ako to spraviť. Najskôr vygenerujeme všetky možné dvojetapové úseky z mesta U, a potom prejdeme všetkých kandidátov pre mesto W, a podľa zapamätaných dvoch možností ľahko zistíme, ktorý je najlepší.

Takto dostávame riešenie s časovou zložitosťou $O(ND^2 + N^2)$ a pamäťovou $O(ND)$.

Aj toto riešenie sa dá ešte približne štyrikrát urýchliť tým, že budeme generovať len tie možnosti, pre ktoré má U menšie číslo ako ostatné tri mestá.

(Navyše vieme jemne upraviť časovú zložitosť na $O(ND^2)$, ak použijeme drobný trik aby sme sa vyhli inicializácii potrebnej pamäte, a následne budeme namiesto všetkých kandidátov pre W prezerať len tých, do ktorých sa aspoň jednou dvojetapovou cestou dá z U dostať. Toto však dĺžku behu nášho programu na najväčšom vstupe výrazne nezmení, preto túto optimalizáciu neimplementujeme.)

Listing programu:

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

int N, M, best=-1, bestm[4];
vector< vector<int> > kam, kolko;

int main() {
  // nacitame vstup
  cin >> N >> M;
  kam.resize(N); kolko.resize(N);
  for (int m=0; m<M; m++) {
    int x,y,h; cin >> x >> y >> h; x--; y--;
    kam[x].push_back(y); kolko[x].push_back(h);
    kam[y].push_back(x); kolko[y].push_back(h);
  }

  for (int u=0; u<N; u++) {
    vector<int> kadial1(N), kolko1(N,-1), kadial2(N), kolko2(N,-1);

    // spracujeme vsetky cesty dlzky 2 z mesta u
```

```
    for (int i=0; i<int(kam[u].size()); i++) {
      int v = kam[u][i];
      if (v<=u) continue;
      for (int j=0; j<int(kam[v].size()); j++) {
        int w = kam[v][j], cena = kolko[u][i] + kolko[v][j];
        if (w<=u) continue;
        if (cena > kolko2[w]) { kolko2[w]=cena; kadial2[w]=v; }
        if (kolko2[w] > kolko1[w]) {
          swap(kadial1[w],kadial2[w]);
          swap(kolko1[w],kolko2[w]);
        }
      }
    }

    // prezrieme vsetkych kandidatov pre mesto w
    for (int w=0; w<N; w++) if (kolko2[w]!=-1) if (kolko1[w]+kolko2[w] > best) {
      best=kolko1[w]+kolko2[w];
      bestm[0]=u; bestm[1]=kadial1[w]; bestm[2]=w; bestm[3]=kadial2[w];
    }
  }

  // vypiseme riesenie
  if (best==-1) cout << "NEEXISTUJE" << endl; else {
    cout << best << endl;
    cout << bestm[3]+1; for (int i=0; i<4; i++) cout << " " << bestm[i]+1;
    cout << endl;
  }
}
```

A-III-5 Robot

Základné riešenie:

Prvým krokom k vyriešeniu úlohy je vedieť simulovať pohyby robota. Stav robota v ľubovoľnom okamihu si vieme popísať štyrmi číslami: dve súradnice políčka, na ktorom stojí, jedno číslo predstavujúce smer, ktorým pozerá, a číslo inštrukcie, ktorú má vykonať ako ďalšiu.

Smery si očíslujeme zaradom, napríklad 0 bude sever, 1 východ, 2 juh a 3 západ. Otočenie doprava teda bude smer zvyšovať a otočenie doľava znižovať o 1 (počítané modulo 4).

Pri úlohách ako je táto je dôležité vyhnúť sa kopírovaniu kusov programu – ľahko tak vznikajú chyby (ktoré sa navyše ťažko hľadajú), keď na niektorom mieste napríklad zabudneme zmeniť $+1$ na -1. A navyše je potom takéto riešenie dlhé a neprehľadné.

Omnoho lepšie je zapísať si na začiatku programu všetko potrebné ako konštanty, a vo zvyšku programu ich len využívať. V našej situácii by tie konštanty mohli vyzerať napríklad takto:

```
// dr[d] = o kolko sa zmeni riadok, ak spravim krok v smere d?
// dc[d] = o kolko sa zmeni stlpec, ak spravim krok v smere d?
int dr[4] = {-1, 0, 1, 0}, dc[4] = {0, 1, 0, -1};
```

O chvíľu sa smelo môžeme pustiť do simulácie pohybu robota. Ale ešte pred tým nám ostáva jedna dôležitá otázka – ako spoznáme, že simulovaný robot nikdy zo stola nepadne? Kedy môžeme prestať simulovať?

Odpoveď na túto otázku sme vlastne uviedli hneď na začiatku tohto riešenia, len sme si ešte neuvedomili, že ju vieme. Zopakujme teda túto myšlienku: Stav robota v ľubovoľnom okamihu si vieme popísať štyrmi číslami. A každé z týchto čísel môže nadobúdať len konečne veľa rôznych hodnôt.

Ak bude robot behať po stole do nekonečna, musí sa teda časom nejaká kombinácia hodnôt zopakovať – inými slovami, robot bude po druhý krát stáť presne na tom istom mieste, pozerať tým istým smerom a na rade bude tá istá inštrukcia.[5]

A naopak, akonáhle sa robot ocitne po druhý krát v presne tom istom stave, je už všetko jasné – dokola bude opakovať postupnosť krokov, ktorá ho do tohto stavu opäť a opäť privedie, a teda (keďže doteraz zo stola nepadol) už nikdy zo stola nepadne.

Stačí teda simulovať pohyb robota až dovtedy, kým buď zo stola nespadne, alebo kým sa neocitne druhýkrát v tom istom stave.

Takto teda dostávame funkčné riešenie: pre každé z $O(WH)$ prázdnych políčok odsimulujeme pohyb robota a výsledok simulácie si zapamätáme. Každá simulácia bude mať najviac $4WHL$ krokov. To preto, že nanajvýš v toľkých rôznych stavoch môže robot byť, a teda sa nanajvýš po $4WHL$ krokoch (podľa Dirichletovho princípu) nejaký stav už musí zopakovať.

Takéto riešenie má teda časovú zložitosť $O(WH \cdot 4WHL) = O(W^2H^2L)$.

Iný spôsob ako jednoduchšie implementovať túto myšlienku: Netreba si pamätať, cez ktoré stavy prechádza aktuálna simulácia, stačí si počítať kroky. Ak odsimulujeme $4WHL$ krokov robota, vieme, že sa zacyklil.

Poznámka: Nie je ľahké zostrojiť vstup, na ktorom by naozaj každá simulácia pohybu robota bola dlhá. Autori úlohy našli konštrukciu, ktorá pre dané N

[5] Na tomto mieste je dobré zdôrazniť, že tým stavom, ktorý sa ako prvý zopakuje, **nemusí** byť začiatočná pozícia. Pohyb robota sa môže zacykliť až po nejakom čase.

zostrojí mapu $N \times N$ a postupnosť príkazov dĺžky N také, že časová zložitosť vyššie uvedeného algoritmu bude priamo úmerná N^4. Takéto vstupy boli medzi testovacími vstupmi.

Lepšie riešenie:

Predchádzajúce riešenie robí kopec zbytočnej práce. Môže sa totiž stať, že roboty, ktoré začínajú na rôznych políčkach, sa časom dostanú do toho istého stavu. Toto zatiaľ naše riešenie nedokázalo zistiť a ani využiť, vždy sme museli robiť celú simuláciu od začiatku až do konca.

Chyba, ktorej sme sa dopustili, bola v tom, že sme po každej simulácii zahodili kopu informácií, ktoré sme počas nej získali. Keď totiž odsimulujeme pohyb robota, dozvedeli sme sa odpoveď nie len pre jeho začiatočný stav – ale aj pre všetky stavy, cez ktoré sme počas simulácie prešli.

Myšlienka nášho vzorového riešenia bude veľmi jednoduchá: Pre každý stav, ktorý sme už niekedy spracovali, si budeme pamätať, koľko krokov z neho ešte robot spraví. Keď teraz ideme spracovať nové začiatočné políčko, môžeme túto informáciu využiť. Simulovať kroky teda budeme len dovtedy, kým sa nedostaneme do známeho stavu. Vtedy namiesto toho, aby sme simulovali ďalej, jednoducho použijeme získanú informáciu.

Čo sme týmto získali? To, že naše vylepšené riešenie už bude každý stav spracúvať len raz. A teda celková časová zložitosť nášho nového riešenia bude priamo úmerná počtu stavov – teda $O(WHL)$.

Vzorové riešenie:

Predchádzajúce riešenie má ešte jeden nedostatok. Bolo by už dostatočne rýchle, problémom je ale, že si pri ňom potrebujeme pamätať riešenie pre každý z $4WHL$ možných stavov. Pre obmedzenia dané v zadaní to je až 500 miliónov hodnôt, a na to by sme potrebovali až dva gigabajty pamäte. A toľko jej k dispozícii nemáme.

Budeme teda musieť nájsť rozumný kompromis a pamätať si len niektoré hodnoty. Dobrý nápad: budeme si pamätať odpovede len pre stavy, kedy je robot na začiatku svojho programu. (Takéto stavy nazvime *zaujímavé*.) Tým sa pamäťové nároky nášho programu znížia na $O(WH)$, čo sa nám už do pamäte pohodlne zmestí. Čo to ale spraví s časovou zložitosťou? Ukážeme, že tá sa nezmení, teda zostane naďalej $O(WHL)$.

Pripomeňme si, ako naše riešenie vyzerá. Postupne budeme spracúvať všetkých WH možných začiatočných stavov robota. Z každého z nich budeme simulovať kroky, až kým sa nedostaneme do situácie, kedy už vieme odpoveď – teda

kým buď simulovaný robot nespadne zo stola, nezacyklí sa, alebo neprídeme do *zaujímavého* stavu, ktorý sme už pred tým spracovali.

Podobne ako v predchádzajúcom riešení si teraz môžeme uvedomiť, že každý z $4WH$ *zaujímavých* stavov budeme spracúvať len raz. No a v tomto prípade na spracovanie *zaujímavého* stavu potrebujeme odsimulovať nanajvýš L krokov, preto celková časová zložitosť nášho riešenia bude naozaj $O(WHL)$.

Poznámka na záver:

Keby sme naše riešenie implementovali pomocou rekurzie, môže sa nám stať nepríjemná vec – hĺbka rekurzie môže byť natoľko veľká, že nám pretečie zásobník. Preto náš program nie je rekurzívny. Namiesto toho jednoducho generuje navštívené stavy do poľa.

Listing programu:

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

int W, H, L;             // parametre zo zadania
char cmd[128];           // postupnost prikazov
char mapa[512][512];     // mapa stola
int answer[512][512][4]; // odpovede: -1: neviem, -2: prave spracuvam, 0: cyklus

struct stav { int r,c,d,l; }; // riadok, stlpec, smer a nasledujuca instrukcia

// konstanty pre pohyb jednotlivymi smermi
int dr[]={-1,0,1,0}, dc[]={0,1,0,-1};

// funkcia wall(stav) vrati true, ak je robot na policku so stenou
bool wall(const stav &current) {
  return (current.r >= 0 && current.r < H)
      && (current.c >= 0 && current.c < W)
      && (mapa[ current.r ][ current.c ] == '#');
}

// funkcia outside(stav) vrati true, ak robot padol zo stola
bool outside(const stav &current) {
  return current.r < 0 || current.r >= H || current.c < 0 || current.c >= W;
}

// funkcia next(stav) vrati nasledujuci stav robota
stav next(const stav &current) {
  stav result;
  // default: suradnice aj smer zostanu, instrukcia stupne o 1
  result.r = current.r;
  result.c = current.c;
```

```
  result.d = current.d;
  result.l = (current.l + 1) % L;

  // rozdiely oproti defaultu: podla prikazu
  switch (cmd[ current.l ]) {
    case 'R': result.d = (current.d + 1) % 4; break;
    case 'L': result.d = (current.d + 3) % 4; break;
    case '+':
      // skusime sa posunut podla aktualneho smeru
      result.r += dr[result.d]; result.c += dc[result.d];
      if (wall(result)) { result.r = current.r; result.c = current.c; }
    break;
    case '-':
      result.r -= dr[result.d]; result.c -= dc[result.d];
      if (wall(result)) { result.r = current.r; result.c = current.c; }
    break;
  }
  return result;
}

int main() {
  // nacitame vstup
  scanf("%d%s%d%d",&L,cmd,&W,&H);
  for (int r=0; r<H; r++) scanf("%s",mapa[r]);
  memset(answer,-1,sizeof(answer));

  // spracujeme vstup
  for (int r=0; r<H; r++) for (int c=0; c<W; c++)
    if (mapa[r][c]=='.') if (answer[r][c][0]==-1) {
      answer[r][c][0] = -2;
      stav S; S.r=r; S.c=c; S.d=0; S.l=0;
      vector<stav> V; V.push_back(S);
      while (1) {
        S = next(S);
        if (outside(S)) {
          // padol zo stola
          int X = V.size();
          for (int i=0; i<X; i++)
            if (V[i].l==0) answer[ V[i].r ][ V[i].c ][ V[i].d ] = X-i;
          break;
        }
        if (S.l==0 && (answer[S.r][S.c][S.d]==-2 || answer[S.r][S.c][S.d]==0)) {
          // zacyklil sa
          int X = V.size();
          for (int i=0; i<X; i++)
            if (V[i].l==0) answer[ V[i].r ][ V[i].c ][ V[i].d ] = 0;
          break;
        }
        if (S.l==0 && answer[S.r][S.c][S.d]>0) {
          // uz vieme po kolkych krokoch padne
```

```
          int X = V.size(), Y = answer[S.r][S.c][S.d];
          for (int i=0; i<X; i++)
            if (V[i].l==0) answer[ V[i].r ][ V[i].c ][ V[i].d ] = X+Y-i;
          break;
        }
        if (S.l==0) answer[S.r][S.c][S.d]=-2;
        V.push_back(S);
      }
    }

  // vypiseme vystup
  int padne = 0;
  for (int r=0; r<H; r++) for (int c=0; c<W; c++) if (answer[r][c][0]>0) padne++;
  printf("%d\n",padne);
  for (int r=0; r<H; r++) {
    for (int c=0; c<W; c++) printf("%s%d",c?" ":"",max(0,answer[r][c][0]));
    printf("\n");
  }
}
```

Výsledky krajských kôl kategórie B

V kategórii B súťaž končí krajským kolom, nižšie uvedené sú výsledky tohto kola v siedmich z ôsmich krajov. (V Žilinskom kraji sa do kategórie B nezapojil žiaden riešiteľ.)

Výsledky neboli koordinované, je teda možné, že v rôznych krajoch boli použité mierne odlišné bodovacie škály.

Bratislavský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Boris Vavrík	2 Gym. Jura Hronca BA	10	7	2	5	24
2. Mariana Phuong	2 Gym. Jura Hronca BA	5	10	5	1	21
3. Juraj Masár	2 Gym. Jura Hronca BA	2	10	2	3	17
Martin Pinter	2 Gym. Jura Hronca BA	0	10	1	6	17
Matej Balog	2 Gym. Grösslingová BA	5	5	2	5	17
6. Richard Trebichavský	1 Gym. Jura Hronca BA	1	10	0	1	12
7. Martin Strapko	2 Gym. Jura Hronca BA	1	7	2	1	11
8. Martin Kovár	2 Gym. Jura Hronca BA	0	8	0	1	9
9. Mojmír Mutný	1 Gym. Jura Hronca BA	1	6	0	1	8
10. Anna Dresslerova	2 Gym. Jura Hronca BA	0	5	1	0	6
Juraj Machac	2 Gym. Jura Hronca BA	0	2	0	4	6
12. Tomas Hruby	2 Gym. Jura Hronca BA	1	1	0	2	4
13. Mária Mrocková	2 Gym. Jura Hronca BA	0			1	1

Košický kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Tomáš Babej	2 Gym. Poštová Košice	3	5	6	1	15
2. Tomáš Livora	2 Gym Javorová Spiš. Nová Ves	4	6	3	1	14
3. Richard Konečný	2 Gym. Šrobárova Košice	2	3	1	4	10
4. Miroslav Riedel	2 Gym. Šrobárova Košice	0	3	0	1	4
5. Miroslava Klučárová	1 Gym Javorová Spiš. Nová Ves	0	0	0	1	1

Nitriansky kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Tomáš Morvay	2 Gym. Golianova Nitra	4	8	6	6	24
2. Michaela Zaťková	2 Gym. Golianova Nitra	3	4	8	4	19
3. Martin Kučera	2 Gym. Golianova Nitra	4	5	8	1	18
4. Viktor Toman	2 Gym. Golianova Nitra	2	7	1	3	13
5. Martin Porubsky	2 Gym. sv. Cyrila a Metoda Nitra	1	7		4	12
6. Andrej Mariš	1 Gym. Piaristická Nitra	1	7	0	1	9
7. Martin Macák	1 Gym. Golianova Nitra	2	1	2	2	7
8. Matúš Balogh	2 Gym. Golianova Nitra	1	3		1	5
9. Lucia Mészárosová	2 Gym. Golianova Nitra	1	0	2	1	4
10. Monika Urbaníková	2 Gym. Golianova Nitra	0	0		3	3
11. Róbert Česár	1 Gym. Golianova Nitra	0	0	2	0	2
12. Juraj Karlubík	1 Gym. Golianova Nitra	0	0	0	1	1
13. Matúš Sitkey	2 Gym. Golianova Nitra	0			0	0

Prešovský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Matúš Nemec	2 Gym. Kukučínova Poprad	2	10	0	2	14
2. Pavol Šatala	1 Gym. Kežmarok	1	7	3	2	13
3. Ivana Krajňáková	2 Gym. Daxnera V.n. Topľou	2	6	0	0	8

Trenčiansky kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Milan Mikuš	1 Gym. Ul. 1. mája Trenčín		9		1	10
2. Martin Betak	2 Gym. Nedožerského Prievidza	2	6		1	9

Trnavský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Radoslav Rábara	2 Gym. Hollého Trnava		5		7	12

Výsledky celoštátneho kola kategórie A

Celoštátne kolo kategórie A sa v 24. ročníku Olympiády v informatike uskutočnilo v dňoch 25. až 28. marca 2009 v Rajeckej Lesnej. Praktické kolo súťažiaci riešili v priestoroch Žilinskej univerzity v Žiline.

	Meno	Ročník, škola	1.	2.	3.	4.	5.	Σ
1.	Fulla Peter	4 SPŠ strojnícka, Sp. N. Ves	10	10	7	15	15	57
2.	Belan Tomáš	sep. ŠPMNDaG, Bratislava	6	10	7	15	15	53
3.	Šrámek Martin	okt. G. Alejová, Košice	5	10	7	15	15	52
4.	Hudec Tobiáš	3 G. Komenského, Partizánske	1	10	7	14	15	47
5.	Herencsár Albert	okt. G. Z. Kodálya, Galanta	5	10	7	15		37
6.	Bačo Ladislav	3 G. Poštová, Košice	4	9	7	11	5	36
7.	Csiba Peter	okt. ŠPMNDaG, Bratislava	6	10	7	0	12	35
8.	Bubnár Michal	okt. G. sv. Fr. z Assisi, Vranov n/T	6	9	5	5	6	31
	Sládek Filip	sep. G. A. Bernoláka, Námestovo	6	7	7	6	5	31
10.	Kováč Jakub	4 G. sv. Cyrila a Metoda, Nitra	8	10	6	5		29
	Petrucha Michal	4 G. Metodova, Bratislava	4	10	7	8		29
	Proksa Ondrej	okt. G. Párovská, Nitra	6	9	7	5	2	29
13.	Dorner Michal	4 G. J. Hronca, Bratislava	3	6	5	14		28
	Kuzma Tomáš	okt. G. Alejová, Košice	6	9	7	6	0	28
15.	Rohár Pavol	3 G. M. R. Štefánika, Košice	6	9	6	4		25
	Baláž Martin	okt. G. Alejová, Košice	5	8	7	0	5	25
17.	Kikta Michal	4 G. D. Tatarku, Poprad	2	9	4	8		23
	Hozza Michal	4 G. J. Hronca, Bratislava	5	7	5	1	5	23
19.	Fekiač Jozef	okt. G. Grösslingová, Bratislava	5	6	9	2		22
	Hozza Ján	sex. G. J. Hronca, Bratislava	5	10	7			22
21.	Cuc Bruno	sep. G. Grösslingová, Bratislava	5	6	6	4		21
	Szabó Erich	4 G. L. Novomeského, Senica	4	6	7		4	21
23.	Kieferová Mária	okt. G. sv. Františka, Žilina	4	9	7	0		20
	Lukča Pavol	sep. G. sv. Fr. z Assisi, Vranov n/T	0	6	5		9	20
25.	Vraník Milan	4 G. J. Hronca, Bratislava	2	6	7	0	4	19
26.	Krejčír Andrej	3 G. Nedožerského, Prievidza	4	7	2		5	18
27.	Klučár Marek	3 G. Javorová, Sp. N. Ves	0	5	9		3	17
28.	Liška Igor	4 G. J. Hronca, Bratislava	0	6	0	5	5	16
29.	Eiben Eduard	4 G. Poštová, Košice	5	4	5		1	15

Výsledky výberového sústredenia

V dňoch 27. apríla až 3. mája 2009 sa v Bratislave konalo výberové sústredenie. Na toto sústredenie boli pozvaní najlepší riešitelia celoštátneho kola OI, kategórie A. Štyria najlepší riešitelia výberového sústredenia majú možnosť reprezentovať Slovensko na Medzinárodnej olympiáde v informatike. Na základe výberového sústredenia taktiež vyberá SK OI reprezentačný tím pre Stredoeurópsku olympiádu v informatike a pre prípravné sústredenia.

V nasledujúcej tabuľke sú postupne uvedené body za celoštátne kolo OI, za „domáce úlohy“ a za sedem súťažných dní výberového sústredenia.

Meno	$\sum$	CK	kód	liah.	po	ut	st	št	pi	so	ne
1. Peter Fulla	664.0	57	3	16.5	64.0	80.0	113.0	106.0	70.0	108.5	46.0
2. Tomáš Belan	522.0	53	0	12.5	48.0	90.5	80.0	80.0	60.5	53.5	44.0
3. Martin Šrámek	453.5	52	2	9.5	39.0	79.0	57.5	68.5	46.0	61.0	39.0
4. Peter Csiba	451.5	35	0	11.5	29.0	87.0	57.5	111.5	11.0	71.0	38.0
5. Tobiáš Hudec	426.5	47	0	2.5	42.5	77.0	85.5	73.5	15.0	45.0	38.5
6. Albert Herencsár	386.0	37	0	11.0	33.0	39.0	78.0	80.0	32.5	45.5	30.0
7. Michal Bubnár	368.5	31	1	18.5	21.0	52.0	34.5	81.0	39.0	60.5	30.0
8. Ladislav Bačo	348.5	36	3	6.0	5.5	38.5	67.0	68.5	32.0	49.0	43.0
9. Michal Petrucha	331.0	29	3	11.0	11.5	72.5	47.0	79.0	11.0	55.0	12.0
10. Filip Sládek	328.5	31	0	15.5	16.5	48.5	58.0	57.0	47.0	29.0	26.0
11. Ján Hozza	279.5	22	3	0.0	13.5	37.5	50.0	49.5	13.0	61.5	29.5
12. Ondrej Proksa	258.5	29	2	16.0	10.0	25.0	46.5	67.5	11.0	29.5	22.0
13. Jakub Kováč	207.5	29	3	13.0	4.0	12.0	60.5	18.0	19.0	31.0	18.0

Slovensko-švajčiarske prípravné sústredenie

Vďaka blízkej spolupráci medzi národnými olympiádami v informatike na Slovensku a vo Švajčiarsku mali vybraní riešitelia KSP a OI možnosť zúčastniť sa prípravného sústredení vo švajčiarskom Davose vo februári 2009. Výsledky sústredenia uvádzame v nasledujúcej tabuľke.

Meno	day 1	day 2	day 3	day 4	$\sum$
1. Peter Fulla	340	278	200	300	1118
2. Tomáš Belan	238	242	140	190	810
3. Martin Šrámek	200	182	140	170	692
4. Isaac Deutsch	220	166	110	180	676
5. Adrian Roos	190	136	160	140	626
6. Daniel Graf	186	182	120	110	598
7. Timon M. Gehr	170	0	160	200	530
8. Simon Laube	184	136	160	20	500
9. Josef Ziegler	180	100	30	160	470
10. Christian Zommerfelds	160	70	120	80	430
11. Tobiáš Hudec	200	50	100	60	410
12. Hans Sjoekvist	116	110	120	50	396
13. Beat Küng	136	50	140	40	366
14. Sämi Grütter	128	0	20	70	218
15. Lorenz Hulfeld	100	40	0	0	140

Česko-poľsko-slovenské prípravné sústredenie

V poradí jedenáste súťažné stretnutie najlepších stredoškolákov z Čiech, Poľska a Slovenska sa uskutočnilo v poľskej Varšave v dňoch 21. až 27. júna 2009. Slovensko tentokrát vyslalo len päťčlennú delegáciu, no jej umiestnenie bolo jedným z najlepších za posledné roky.

Meno, krajina		$\sum$	day1	day2	day3	day4
1. Jakub Pachocki	POL	1143	243	300	300	300
2. Peter Fulla	SVK	949	247	292	240	170
3. Jakub Adamek	POL	855	238	300	232	85
4. Jarosław Błasiok	POL	708	240	300	53	115
5. Tomáš Belan	SVK	694	189	260	185	60
6. Peter Csiba	SVK	665	239	151	140	135
7. Adrian Jaskółka	POL	642	278	199	150	15
8. Martin Šrámek	SVK	600	134	271	150	45
9. Vlastimil Dort	CZE	574	114	189	236	35
10. Ján Hozza	SVK	552	225	148	154	25
11. Adam Karczmarz	POL	552	110	300	142	0
12. Tomasz Kociumaka	POL	489	0	0	194	295
13. Hynek Jemelik	CZE	455	107	67	141	140
14. Martin Patera	CZE	310	78	70	97	65
15. Jan Polášek	CZE	291	110	0	81	100
16. Jakub Oćwieja	POL	248	0	0	83	165
17. Karel Tesař	CZE	153	10	100	28	15
18. Lukáš Kripner	CZE	42	10	12	20	0

Stredoeurópska olympiáda v informatike

V roku 2009 sa Stredoeurópska olympiáda v informatike (CEOI) konala v dňoch 8. až 14. júla v rumunskom meste Tîrgu Mureş. Súťaže sa zúčastnili stredoškoláci z jedenástich krajín: okrem tradičných siedmich (Českej republiky, Chorvátska, Maďarska, Nemecka, Poľska, Rumunska a Slovenska) tentokrát organizátori pozvali tímy z Moldavska, Spojených štátov amerických, Srbska a Švajčiarska.

Našu delegáciu tentokrát tvorili doc. RNDr. Gabriela Andrejková, CSc., RNDr. Rastislav Krivoš-Belluš a Mgr. Ján Katrenič, všetci z ÚI PF UPJŠ v Košiciach.

Súťažiaci reprezentujúci Slovensko boli Tomáš Belan, Peter Csiba (obaja zo Školy pre mim. nadané deti, Bratislava), Michal Bubnár (Gym. sv. Františka z Assisi, Vranov nad Topľou) a Albert Herencsár (Gym. Z. Kodálya, Galanta). Výsledky našich súťažiacich uvádzame zhrnuté v tabuľke.

	Meno	1. deň			2. deň			$\sum$	medaila
17.	Belan Tomáš	10	20	–	100	10	30	170	bronzová
27.	Herencsár Albert	0	20	40	40	10	20	130	bronzová
	Csiba Peter							100	
	Bubnár Michal							100	

(Presné body Petra Csibu a Michala Bubnára sa nezachovali.)

Medzinárodná olympiáda v informatike

V roku 2009 sa Medzinárodná olympiáda v informatike (IOI) po dvadsiatich rokoch vrátila späť do krajiny, kde začínala – do Bulharska.

V roku 1989 sa v bulharskom meste Pravetz konala úplne prvá oficiálna IOI. Zúčastnilo sa jej 13 krajín: Bulharsko, Československo, Čina, Grécko, Juhoslávia, Kuba, Maďarsko, Nemecká demokratická republika, Nemecká spolková republika, Poľsko, Sovietsky zväz, Vietnam a Zimbabwe. Celkový počet súťažiacich na prvej IOI bol 46. Za zmienku stojí, že jedným zo zlatých medailistov na prvej IOI bol aj Slovák Igor Malý so ziskom 95 zo 100 možných bodov. Vedúcim československej delegácie na prvej IOI bol RNDr. Ondrej Demáček.

Za 20 rokov sa mnohé zmenilo. V dňoch 8. až 15. augusta 2009 do bulharského Plovdivu prišlo 311 súťažiacich z 80 krajín sveta. Súťaž už nepozostávala z jedinej súťažnej úlohy vyhodnocovanej odbornou komisiou, ale z dvoch súťažných dní, pričom v každý deň súťažiaci riešili štyri rôzne náročné úlohy. Všetky riešenia úloh boli po skončení súťaže automaticky otestované na vopred pripravených sadách testovacích dát.

Ako lídri zastupujúci našu krajinu pri hlasovaniach sa tejto IOI zúčastnili RNDr. Andrej Blaho a Mgr. Juliana Lipková z FMFI UK v Bratislave. Ako člen medzinárodnej odbornej komisie (ISC) sa tejto IOI zúčastnil RNDr. Michal Forišek, PhD. z FMFI UK v Bratislave.

Našu krajinu tento rok reprezentovala táto štvorica stredoškolákov: Tomáš Belan a Peter Csiba z Školy pre mimoriadne nadané deti, Bratislava, Peter Fulla z SPŠ Spišská Nová Ves a Martin Šrámek z Gymnázia Alejová, Košice. Výsledky našich súťažiacich uvádzame zhrnuté v tabuľke.

Meno	1. deň				2. deň				$\sum$	medaila
Tomáš Belan	21	49	100	100	100	76	45	62	553	striebro
Peter Fulla	20	25	100	100	100	100	40	60	545	striebro
Peter Csiba	21	36	100	100	100	100	13	17	487	bronz
Martin Šrámek	10	10	100	100	100	17	12	0	349	

Zadania prvého súťažného dňa

Plovdivská olympiáda v informatike

Plovdivská olympiáda v informatike mala nasledovné, mierne neobvyklé pravidlá: N súťažiacich riešilo T rôznych úloh. Pred súťažou každý súťažiaci dostal unikátne ID číslo z rozsahu od 1 do N.

Každá úloha bola testovaná len na jednom vstupe. Preto každú úlohu mohol daný súťažiaci buď vyriešiť, alebo nie. Čiastočne úspešné riešenie nebolo možné. Počet bodov, ktoré boli priradené za vyriešenie úlohy, bol určený po súťaži. Bol rovný počtu súťažiacich, ktorí danú úlohu nevyriešili. Celkový počet bodov, ktoré súťažiaci získal, bol rovný súčtu bodov za úlohy, ktoré vyriešil.

Finálne výsledky sú v prvom rade zoradené podľa bodov (od najviac po najmenej). V prípade, že dvaja súťažiaci majú rovnaký počet bodov, tak súťažiaci, ktorý vyriešil viac úloh, bude vo výsledkovej listine pred tým, ktorý vyriešil menej úloh. Pokiaľ súťažiaci majú rovnako veľa bodov a vyriešili rovnaký počet úloh, tak budú zoradení podľa ID (od najnižšieho po najvyššie).

Súťažiaci Georg, ktorého ID je P, je zmätený z komplikovaných pravidiel prideľovania bodov. Zúfalo potrebuje vašu pomoc.

Súťažná úloha:

Napíš program, ktorý dostane informácie o tom, ktorý súťažiaci ktoré úlohy vyriešil, a následne zistí, koľko bodov Georg získal a na ktorej pozícii v konečnom poradí sa umiestnil.

Ohraničenia:

$1 \leq N \leq 2\,000$	Počet súťažiacich
$1 \leq T \leq 2\,000$	Počet úloh
$1 \leq P \leq N$	Georgove ID

Vstup:

Tvoj program má zo štandardného vstupu prečítať nasledujúce dáta:

Prvý riadok obsahuje celé čísla N, T a P, oddelené medzerami.

Nasledujúcich N riadkov popisuje, ktorú úlohu, ktorý súťažiaci vyriešil. K-ty riadok popisuje, ktoré úlohy vyriešil súťažiaci s ID k. Každý z týchto riadkov obsahuje T čísel oddelených medzerou. Prvé z týchto čísel vyjadruje, či súťažiaci vyriešil prvú úlohu. Druhé, či vyriešil druhú, atď. Každé z týchto T čísel je buď 0 (nevyriešená úloha) alebo 1 (vyriešená úloha).

Výstup:

Tvoj program má vypísať jeden riadok, ktorý obsahuje 2 čísla oddelené medzerou. Prvé číslo vyjadruje počet bodov, ktoré Georg získal a druhé Georgove umiestnenie. Umiestnenie je číslo od 1 do N (1 vyjadruje súťažiaceho, ktorý skončil najlepšie – čo je jeden z tých, čo mali najvyššie skóre).

Bodovanie:

Vo vstupoch za 35 bodov je zaručené, že žiadny iný súťažiaci nebude mať rovnako veľa bodov ako Georg.

Príklad:

Vstup	Výstup
5 3 2 0 0 1 1 1 0 1 0 0 1 1 0 1 1 0	3 2

Prvý problém nevyriešil len jeden súťažiaci, preto je zaň 1 bod. Druhý problém je za 2 body, tretí za 4. Prvý súťažiaci získal 4 body; druhý (Georg), štvrtý a piaty získali 3 body; tretí získal 1 bod. Keďže Georg a ďalší dvaja majú rovnaký počet bodov a navyše aj rovnaký počet vyriešených úloh, tak rozhoduje ID. To má Georg najnižšie. Preto sa umiestnil ako druhý za súťažiacim s ID 1.

Hrozienka

Bonny, známa to plovdivská výrobkyňa čokolád, potrebuje narezať tabličku hrozienkovej čokolády. Čokoláda je obdĺžnikového tvaru a skladá sa z $N \times M$ identických kociek. Každá kocka obsahuje niekoľko hrozienok (hrozienka neležia na hraniciach medzi dvoma kockami).

Na začiatku je čokoláda v jednom celku. Bonny potrebuje túto čokoládu postupne narezať na NM jednotlivých kociek. Kedže ale Bonny je veľmi zaneprázdnená, poprosila o narezanie svojho asistenta Čierneho Petra. Peter však robí len priame rezy (rez musí viesť pozdĺž hrany a oba jeho konce musia byť na obvode). Za každý takýto rez mu Bonny musí zaplatiť. Kedže Bonny nemá akurát peniaze poruke, dohodli sa s Petrom, že bude vyplatený v zvyšných hrozienkach, ktoré Bonny zostali. Ich dohoda je nasledovná: zakaždým, keď Peter

dostane kus čokolády, ktorý má rozrezať na dve časti, zaplatí mu Bonny toľko hrozienok, koľko ich je na dotyčnom kuse čokolády.

Bonny by chcela dať Petrovi čo najmenej hrozienok. Vie počty hrozienok na jednotlivých kockách tabličky. Na základe toho si môže zvoliť, v akom poradí bude Petrovi dávať kúsky čokolády a ako mu ich prikáže rezať. (Bonny si môže vždy povedať pozíciu a smer rezu, ktorý má Peter spraviť.) Zistite, koľko najmenej hrozienok musí Bonny Petrovi dokopy zaplatiť.

Súťažná úloha:

Napíš program, ktorý pre zadané počty hrozienok na jednotlivých kockách tabličky čokolády vypíše minimálne množstvo hrozienok, ktorými musí Bonny zaplatiť Petrovi za rezanie.

Ohraničenia:

$1 \leq N, M \leq 50$	Veľkosti strán tabličky
$1 \leq R_{K,P} \leq 1\,000$	Počet hrozienok v K-tom riadku a P-tom stĺpci

Vstup:

Tvoj program má načítať zo štandardného vstupu nasledujúce dáta:

Prvý riadok vstupu obsahuje dve celé čísla N a M oddelené medzerou.

Nasledujúcich N riadkov obsahuje počty hrozienok, ktoré sú na každej kocke čokolády. K-ty riadok z týchto N riadkov obsahuje popis K-teho riadku tabličky čokolády. Obsahuje M celých čísel oddelených jednou medzerou. Tieto čísla popisujú zľava doprava jednotlivé kocky daného riadku čokolády. P-te číslo v K-tom riadku (z týchto N riadkov) je počet hrozienok, ktoré sú v kocke v K-tom riadku a P-tom stĺpci mriežky.

Výstup:

Na štandardný výstup vypíš jeden riadok obsahujúci jedno celé číslo – minimálny počet hrozienok, ktoré musí Bonny dať Čiernemu Petrovi na rozrezanie tabličky na kocky.

Bodovanie:

Vo vstupoch v hodnote 25 bodov čísla N a M nepresiahnu hodnotu 7.

Príklad:

Vstup	Výstup
2 3 2 7 5 1 9 5	77

Jeden možný spôsob (z viacerých) ako dosiahnuť cenu 77 je nasledovný:

Pri prvom rezaní, ktoré Bonny požaduje, Peter oddelí tretí stĺpec od zvyšku čokolády, čo bude stáť 29 hrozienok.

Potom Bonny dá narezať menší z dvoch kúskov (každý obsahuje 5 hrozienok) na dve časti za ďalších 10 hrozienok.

Následne Bonny nechá Petrovi najvačší ostávajúci kus (obsahuje kocky s 2, 7, 1 a 9 hrozienkami). Bonny požiada Petra, aby ho narezal horizontálne, za čo zaplatí 19 hrozienok.

Potom Bonny dá Petrovi narezať ľavý horný kúsok a zaplatí 9 hrozninok. Na záver Bonny požiada Petra aby rozdelil zvyšný ľavý dolný kus zaplatiac 10 hrozienok.

Celková cena, ktorú Bonny zaplatí je $29 + 10 + 19 + 9 + 10 = 77$ hrozienok. Žiadne iné rezanie tejto tabličky čokolády na 6 kociek nemá nižšiu celkovú sumu.

Nájom robotníkov

Potrebuješ si prenajať robotníkov na stavebný projekt. Na tento džob sa prihlásilo N kandidátov, ktorí sú očíslovaní od 1 do N, vrátane. Pre k-teho kandidáta platí: ak je prijatý, musí dostať plat minimálne S_k dolárov. Tiež poznáme kvalifikačnú úroveň každého kandidáta: kandidát k má kvalifikáciu Q_k. Pravidlá stavebného priemyslu určujú, že robotníci musia dostať platy priamo úmerné ich kvalifikácii. Napríklad, ak zamestnáte dvoch robotníkov A a B a pritom platí $Q_A = 3 \cdot Q_B$, potom musí byť plat robotníka A presne trikrát vyšší ako plat robotníka B. Robotníkov môžeš platiť aj neceločíselnými sumami dolárov, výplata dokonca môže obsahovať aj nekonečný desatinný zápis, napr. 1/3 alebo 1/6 doláru.

Na zaplatenie všetkých najatých robotníkov máš k dispozícii W dolárov. Chcel by ste najať čo najviac robotníkov. Ty rozhodneš, koho si najmeš a koľko mu zaplatíš, len to nesmie byť menej ako jeho minimum a zároveň musia byť splnené aj pravidlá stavebného priemyslu, t. j. platy všetkých robotníkov musia byť v rovnakom pomere ako ich kvalifikácie.

Pri tvojom projekte ti vôbec nezáleží na kvalifikačnej úrovni robotníkov. Preto sa zaujímaš len o čo najväčší počet prenajatých robotníkov bez ohľadu na ich kvalifikáciu. Ak ale existuje viac možností, ako sa dá toto maximum robotníkov dosiahnuť, tak si vyberieš tú, pri ktorej im dokopy treba zaplatíť najmenej. Ak aj takýchto najlacnejších riešení existuje viac, tak je jedno, ktoré z nich si vyberieš.

Súťažná úloha:

Napíš program, ktorý pre každého kandidáta dostane jeho platové minimum a jeho kvalifikačnú úroveň a tiež dostane celkovú sumu peňazí ma ich platy. Program potom musí rozhodnúť, ktorých kandidátov treba prijať. Treba ich prijať najviac ako sa len dá, a malo by ťa to dokopy stáť čo najmenej. Pritom treba dodržať vyššie uvedené priemyselné pravidlá.

Ohraničenia:

$1 \leq N \leq 500\,000$	počet kandidátov
$1 \leq S_k \leq 20\,000$	minimálny plat k-teho robotníka
$1 \leq Q_k \leq 20\,000$	kvalifikačná úroveň k-teho robotníka
$1 \leq W \leq 10\,000\,000\,000$	celková suma peňazí na platy

Vstup:

Tvoj program by mal čítať zo štandardného vstupu nasledovné dáta:

Prvý riadok obsahuje celé čísla N a W oddelené medzerou. Nasledujúcich N riadkov popisuje kandidátov, každého kandidáta jedným riadkom. Presnejšie, k-ty riadok popisuje kandidáta s číslom k a obsahuje celé čísla S_k a Q_k oddelené medzerou.

Výstup:

Tvoj program má vypísať na štandardný výstup tieto dáta:

V prvom riadku musí byť jedno celé číslo H – počet najatých robotníkov. V nasledovných H riadkoch musí byť zoznam čísel najatých kandidátov (všetky čísla musia byť medzi 1 a N a musia byť navzájom rôzne), jedno v každom riadku, v ľubovoľnom poradí.

Bodovanie:

Za každý testovací vstup získate plný počet bodov, ak výber kandidátov spĺňa všetky kritériá a tiež obmedzenia. Ak je na výstupe správny prvý riadok (t.j. optimálna hodnota H), tak napriek chybnému zvyšku získavate 50% bodov za tento testovací vstup. Toto by sa týkalo aj situácie, ak by výstupný súbor nebol správne naformátovaný, ale prvý riadok by bol správny.

Za testy, v ktorých N nebude väčšie ako 5 000, sa dá získať spolu maximálne 50 bodov.

Príklady:

Vstup	Výstup
4 100 5 1000 10 100 8 10 20 1	2 2 3

Jediný spôsob, ako môžeš najať dvoch robotníkov a pritom dodržať podmienky, je najať robotníkov 2 a 3 a zaplatiť im 80 a 8 dolárov. Toto sa zmestí do 100-dolárového rozpočtu.

Vstup	Výstup
3 4 1 2 1 3 1 3	3 1 2 3

Tu je možné najať všetkých troch robotníkov. Zaplatíš 1 dolár prvému a po 1.50 dolárov obom zvyšným robotníkom, čím presne minieš všetky 4 doláre.

Vstup	Výstup
3 40 10 1 10 2 10 3	2 2 3

Nemôžeme najať všetkých troch robotníkov, to by stálo až 60 dolárov. Vieme ale najať ľubovoľných dvoch. Najlepši je vybrať robotníkov 2 a 3, keďže ich súčet platov bude najmenší v porovnaní s ďalšími dvoma možnosťami. V tomto prípade treba zaplatiť 10 dolárov robotníkovi 2 a 15 dolárov robotníkovi 3, čo je spolu 25 dolárov. Ak by sme najali robotníkov 1 a 2, stálo by nás to aspoň $10 + 20 = 30$ *dolárov, a na najatie 1 a 3 potrebujeme* $10 + 30 = 40$ *dolárov.*

Lukostreľba

Lukostrelecký turnaj v Dolných Kočkovciach má nasledujúce pravidlá: Strieľa sa na N terčov, ktoré sú v rade vedľa seba a očíslované od 1 do N (v poradí

zľava doprava – najľavejší má číslo 1, najpravejší N). V turnaji súťaží $2N$ lukostrelcov. Turnaj trvá R kôl, pričom kôl je aspoň toľko ako súťažiacich ($R \geq 2N$).

Každé kolo turnaja funguje nasledovne: Na každom terči súťažia dvaja súťažiaci. Z tejto súťaže jeden vyjde ako víťaz a druhý ako porazený. Potom sa súťažiaci preusporiadajú nasledovne: Víťazi na terčoch 2 až N sa presunú o jeden terč doľava (t.j. víťaz na terči 2 sa presunie k terču 1, ..., víťaz na terči N k terču $N-1$). Porazení na terčoch 2 až N, ako aj víťaz na terči 1, zostavajú na svojich miestach. Porazený na terčí 1 sa presunie k terču N.

V Dolných Kočkovciach to funguje občas trochu na hlavu. Ty si síce prišiel načas, ale ostatných $2N-1$ lukostrelcov prišlo ešte skôr a už stoja v rade. Jediné, čo teraz môžeš spraviť, je vopchať sa niekam do radu medzi nich. Akonáhle zaujmeš svoje miesto, tak sa tí dvaja súťažiaci, čo budú stáť najviac vľavo v rade, presunú k terču 1, ďalší dvaja k terču 2, ..., a tí najviac napravo dostanú terč N a začne turnaj.

O všetkých súťažiacich (vrátane teba) sú dobre známe ich schopnosti a na ich základe im bolo priradené jedno číslo, ktoré je tým menšie, čím je strelec lepší (ďalej len „rank“). Vieš, že žiadni dvaja súťažiaci nemajú rovnaký rank. A taktiež vieš, že keď dvaja strelci súťažia, tak ten s nižším rankom vždy vyhrá.

Na základe týchto znalostí sa chceš do radu postaviť tak, aby si po skončení súťaže skončil čo najviac naľavo. Pokiaľ je viac možností ako to dosiahnuť, tak preferuješ tú, pri ktorej začínaš najviac vpravo.

Súťažná úloha:

Napíš program, ktorý dostane ranky strelcov (vrátane vás) a poradie ostatných súťažiacich v rade a zistí, kam sa máš postaviť, aby si dosiahol cieľ popísaný vyššie.

Ohraničenia:

$1 \leq N \leq 200\,000$	Počet terčov (takisto polovica počtu súťažiacich)
$2N \leq R \leq 1\,000\,000\,000$	Počet úloh
$1 \leq S_k \leq 2N$	Rank k-teho súťažiaceho

Vstup:

Tvoj program má zo štandardného vstupu prečítať nasledujúce dáta:

Prvý riadok obsahuje dve celé čísla N a R, oddelené medzerou.

Druhý riadok obsahuje ranky strelcov. Prvý z nich je tvoj rank. Nasledujú ranky ostatných strelcov v poradí, v akom sa zoradili pred tvojím príchodom (zľava doprava). Každý rank je číslo medzi 1 a $2N$. Rank 1 je najlepší a rank $2N$ najhorší. Žiadni dvaja súťažiaci nemajú rovnaký rank.

Výstup:

Tvoj program má vypísať jeden riadok s číslom od 1 do N – číslo terča, na ktorom máš začínať turnaj.

Bodovanie:

V testoch za 60 bodov N nepresiahne 5000. V testoch za 20 bodov N nepresiahne 200.

Príklad:

Vstup	Výstup
4 8 7 4 2 6 5 8 1 3	3

Si druhý najhorší. Pokiaľ začneš na terči 1, pôjdeš na terč 4 a tam zostaneš do konca. Pokiaľ pôjdeš na terč 2 alebo 4, tak tam zostaneš celý turnaj. Pokiaľ začneš na terči 3, tak porazíš najhoršieho strelca a presunieš sa na terč 2 a zostaneš tam.

Vstup	Výstup
4 9 2 1 5 8 3 4 7 6	2

Si druhý najlepší. Najlepší je na terči 1 a tam aj zostane. Takže nezávisle na tom, kde začneš, sa budeš presúvať z terča, kde začneš, cez všetky terče dookola. Aby si sa dostal na terč 1 po 9 kolách, musíš začať na terči 2.

Zadania druhého súťažného dňa

Garáž

Parkovacia garáž má N parkovacích miest a tieto sú očíslované od 1 do N. Garáž sa otvára ráno prázdna a počas celého dňa pracuje podľa takýchto pravidiel: Vždy, keď nejaké auto príde a chce zaparkovať v garáži, zodpovedný pracovník skontroluje, či je nejaké parkovacie miesto voľné. Ak nie je, auto musí čakať pred vjazdom, kým sa nejaké miesto neuvoľní. Ak je nejaké miesto voľné, auto nečaká a hneď zaparkuje. Ak je voľných viac parkovacích miest, auto zaparkuje na voľnom mieste s najmenším číslom. Ak počas toho, ako pred

garážou čaká nejaké auto, prídu ďalšie autá, tak sa tieto zaraďujú za sebou do radu presne v tom poradí, v akom prišli. Neskôr, keď sa nejaké miesto uvoľní, prvé auto čakajúce v rade (t. j. to, ktoré čaká najdlhšie) na ňom zaparkuje.

Cena parkovného v dolároch je rovná váhe vozidla v kilogramoch, vynásobenej parkovacím poplatkom pre dotyčné parkovacie miesto. Cena nezávisí od dĺžky parkovania v garáži.

Správca garáže vie, že dnes príde zaparkovať M áut. Tiež pozná presné poradie ich príchodov a odchodov. Pomôžte mu vypočítať, koľko dolárov dnes zarobí.

Súťažná úloha:

Napíš program, ktorý dostane na vstupe poplatky pre všetky parkovacie miesta, váhy všetkých áut a ich poradie príchodu a odchodu a na základe tohto vypočíta celkový dnešný príjem garaže v dolároch.

Ohraničenia:

$1 \leq N \leq 100$	počet parkovacích miest
$1 \leq M \leq 2\,000$	počet áut
$1 \leq R_s \leq 100$	parkovací poplatok pre s-té miesto (doláre/kg)
$1 \leq W_k \leq 10\,000$	váha k-teho auta (kg)

Vstup:

Tvoj program bude čítať zo štandardého vstupu nasledovné dáta:

Prvý riadok obsahuje dve celé čísla N a M, ktoré sú oddelené medzerou. Autá sú očíslované od 1 do M bez nejakého špeciálneho významu ich poradia.

Druhý riadok obsahuje N čísel $R_1, \ldots, R_N$ udávajúcich poplatky za parkovacie miesta v dolároch za kilogram. Tretí riadok obsahuje M čísel $W_1, \ldots, W_M$ udávajúcich váhy áut v kilogramoch.

Posledný riadok popisuje príchody a odchody všetkých áut v ich chronologickom poradí. Kladné celé číslo i označuje, že auto s číslom i prišlo do garáže. Záporné celé číslo $-i$ označuje, že auto s číslom i odišlo z garáže. Môžete predpokladať, že žiadne auto neodíde z garáže skôr ako prišlo. Tiež možete predpokladať, že každé číslo auta (od 1 do M) sa v tejto postupnosti objaví presne dvakrát: raz keď príde a druhýkrát keď odíde. Navyše vždy bude platiť, že žiadne auto neodíde skôr, ako stihlo zaparkovať v garáži (t. j. neodíde, pokým ešte čaká v rade).

Výstup:

Tvoj program by mal na štandardný výstup do jediného riadku vypísať jedno celé číslo: celkovú sumu v dolároch, ktorú dnes zarobí správca garáže.

Bodovanie:

V testoch za 40 bodov bude každé prichádzajúce auto mať k dispozícii aspoň jedno voľné parkovacie miesto. V týchto prípadoch sa teda nebude vytvárať rad áut čakajúcich na parkovacie miesto.

Príklady:

Vstup

```
3 4
2 3 5
200 100 300 800
3 2 -3 1 4 -4 -2 -1
```

Výstup

```
5300
```

– *Auto s číslom 3 ide na parkovacie miesto číslo 1 a zaplatí* $300 \cdot 2 = 600$ *dolárov.*
– *Auto s číslom 2 ide na parkovacie miesto číslo 2 a zaplatí* $100 \cdot 3 = 300$ *dolárov.*
– *Auto 1 ide na miesto 1 (ktoré uvoľnilo auto 3) a zaplatí* $200 \cdot 2 = 400$ *dolárov.*
– *Auto 4 ide na miesto 3 (posledné voľné) a zaplatí* $800 \cdot 5 = 4000$ *dolárov.*

Vstup

```
2 4
5 2
100 500 1000 2000
3 1 2 4 -1 -3 -2 -4
```

Výstup

```
16200
```

– *Auto 3 ide na miesto 1 a zaplatí* $1000 \cdot 5 = 5000$ *dolárov.*
– *Auto 1 ide na miesto 2 a zaplatí* $100 \cdot 2 = 200$ *dolárov.*
– *Prišlo auto 2 a musí čakať pri vstupe do garáže.*
– *Prišlo auto 4 a musí čakať pri vstupe do garáže za autom 2.*
– *Keď auto 1 odišlo, zaparkovalo na mieste 2 auto 2 za* $500 \cdot 2 = 1000$ *dolárov.*
– *Keď auto 3 odišlo, zaparkovalo na mieste 1 auto 4 za* $2000 \cdot 5 = 10000$ *dolárov.*

Medveď

Macko Pú našiel v lese úžasné miesto. Na tomto mieste si včely uchovávali med, ktorý sa im nechcelo držať v úľoch. Nanešťastie, keď si pochutnával nad úžasnou žranicou, tak ho uvidela jedna včela. Tá malá sviňa okamžite spustila poplach. Macko Pú je skusený a vie, že včely okamžite začnú vyliezať z úľov a rozptýlia sa po okolí (aby mali, čo najväčšiu šancu ho chytiť). A vie, že by mal utekať domov (tam sú ňanho včely prikrátke). Ale taký skúsený vyžierač

úľov tiež vie, že nemusí vždy utekať okamžite a môže si ešte chvíľu vychutnávať med. Len nevie presne koľko. Pomôžte mu nájsť ten posledný možný moment, kedy môže odísť.

Les si môžeme predstaviť ako štvorcovú mriežku s rozmermi $N \times N$ políčok. Hrany políčok sú rovnobežné so smermi sever-juh, východ-západ. Na každom políčku je buď strom, úľ, tráva alebo Púou dom. Dva políčka sa považujú za susedne, ak susedia hranou (t.j. keď je jedno na východ, sever, západ, alebo juh od druhého, nie diagonálne). Každý krok Macka Pú smeruje z nejakého políčka na jeho susedné políčko. Môže chodiť po tráve, nemôže chodiť cez stromy a úle a môže urobiť najviac S krokov za minútu.

Keď včely spustili poplach, tak Pú stojí na trávnatom políčku s medom a včely sú na políčkach, ktoré obsahujú úle (tých môže byť aj viac). Počas každej minúty sa stanú nasledovné veci:

Ak sa Pú stále napcháva medom, tak sa môže rozhodnúť, či v tom bude pokračovať alebo odíde. Ak pokračuje, tak sa nepohne celú minútu. Ináč okamžite odchádza a urobí nanajvýš S krokov cez les. Pú si nemôže zobrať med so sebou a akonáhle sa pohne, nemôže jesť znovu.

V okamihu, keď uplynie celá minúta (počas ktorej Pú jedol alebo sa hýbal), sa včely rozptýlia na ďalšie políčka pokryté trávou. Presnejšie roj sa rozptýli na každé políčko, ktoré susedí s políčkom, ktoré už obsahuje včely. Navyše ak sú už na políčku nejaké včely, tak to políčko už bude navždy obsahovať včely (roj sa nepresúva, ale rozrastá).

Inými slovami, roj sa rozptyľuje takto: Keď zaznie alarm, roj okupuje tie políčka, kde sú úle. Na konci prvej minúty okupuje aj všetky trávnaté políčka, ktoré susedia s úľmi (a aj tie, kde boli úle). A tak ďalej. Po dostatočne dlhom čase budú včely okupovať všetky dosiahnuteľné trávnaté políčka v lese.

Ani Pú ani včely nemôžu vyliezť z lesa. Tiež, podľa predchádzajúcich pravidiel, Pú vždy bude jesť celočíselný počet minút.

Včely chytia Púa, ak sa v nejakom okamihu Pú ocitne na políčku, kde sú aj včely.

Súťažná úloha:

Napíš program, ktorý dostane mapu lesa a zistí maximálny počet minút, ktoré môže Pu pokračovať v jedení a stále zmiznúť domov bez toho, aby ho chytili včely.

Ohraničenia:

$1 \leq N \leq 800$	Veľkosť (bočnej strany) mapy lesa
$1 \leq S \leq 1\,000$	Maximálny počet krokov, ktoré Pú spraví za minútu.

Vstup:

Tvoj program má prečítať zo štandardného vstupu nasledujúce dáta:

Prvý riadok obsahuje čísla N a S oddelené medzerou. Ďalších N riadkov popisuje mapu lesa. Každý z týchto riadkov obsahuje N znakov (každý znak reprezentuje jedno políčko). Znaky a ich význam sú nasledovné: `T` značí strom, `G` trávu, `H` úľ, `M` pôvodnú pozíciu Púa (kde sa napcháva medom) a `D` Púov dom.

Je zaručené, že mapa bude obsahovať práve jedno písmeno `M`, práve jedno písmeno `D` a aspoň jedno písmeno `H`. Je zaručené, že existuje cesta (po tráve) medzi miestom, kde je med a Púovym domom, a tiež že existuje cesta medzi miestom, kde je med, a aspoň jedným úľom. (Cesta nemusí obsahovať ani jedno trávnaté políčko, t. j. aj úľ, aj dom môžu priamo susediť s miestom, kde je med.) Včely nemôžu lietať cez miesto, kde sa nachádza Púov dom.

Výstup:

Tvoj program má vypísať jeden riadok obsahujúci jedno celé číslo: maximálny počet minút, ktoré Pú môže jesť med a stále sa dostať bezpečne domov. Ak Pú nemá šancu dôjsť domov bez toho, aby ho chytili včely, vypíšte -1.

Bodovanie:

V testoch, ktoré budú ohodnotené 40 bodmi, N nepresiahne 60.

Príklady:

Vstup

```
7 3
TTTTTTT
TGGGGGT
TGGGGGT
MGGGGGD
TGGGGGT
TGGGGGT
THHHHHT
```

Výstup

```
1
```

Po jednej minúte napchávania Pú pôjde rovno doprava a bude doma za 2 minúty bez toho, aby ho ohrozili včely.

Vstup

```
7 3
TTTTTTT
TGGGGGT
TGGGGGT
MGGGGGD
TGGGGGT
TGGGGGT
TGHHGGT
```

Výstup

```
2
```

Po dvoch minútach napchávania môže Pú urobiť kroky →↑→ počas tretej minúty, potom kroky →→→ počas štvrtej minúty a kroky ↓→ počas piatej minúty.

Regióny

Regionálna rozvojová agentúra spojených národov (UNRDA) má dobre navrhnutú organizačnú štruktúru. Agentúra má celkovo N zamestnancov, každý z nich pochádza z jedného z R geograficky rôznych regiónov sveta. Zamestnanci sú očíslovaní od 1 do N v poradí podľa veku, s tým, že zamestnanec číslo 1, ktorým je predseda, je najstarší. Regióny sú očíslované od 1 do R pričom samotné očíslovanie nemá žiaden špeciálny význam. Každý zamestnanec okrem predsedu má svojho jediného priameho nadriadeného. Nadriadený je vždy starší ako sú zamestnanci, ktorých riadi.

Hovoríme, že zamestnanec A je manažérom zamestnanca B vtedy a len vtedy, ak A je buď priamym nadriadeným zamestanca B, alebo A je manažérom priameho nadriadeného zamestnanca B. To znamená napríklad aj to, že predseda je manažérom všetkých ostatných zamestnancov. A tiež je zrejmé, že žiadni dvaja zamestnanci si nemôžu byť navzájom manažérmi.

Žiaľ, v ostatnom čase dostal Vyšetrovací úrad spojených národov (UNBI) niekoľko sťažnosti na to, že UNRDA má nevyváženú organizačnú štruktúru a niektoré regióny sveta zvýhodňuje viac ako iné. UNBI by kvôli vyšetreniu obvinení chcelo navrhnúť taký počítačový systém, ktorý, keď dostane štruktúru nadriadených z UNRDA, tak bude schopný odpovedať na otázky v tvare: pre dané dva rozličné regióny r_1 a r_2 spočítajte, koľko existuje v agentúre takých dvojíc zamestnancov e_1 a e_2 takých, že zamestnanec e_1 pochádza z regiónu r_1, zamestnanec e_2 pochádza z regiónu r_2 a zároveň e_1 je manažérom zamestnancovi e_2. Každá otázka má dva parametre: regióny r_1 a r_2. Odpoveďou je vždy jediné celé číslo: počet rôznych dvojíc (e_1, e_2), ktoré vyhovujú horeuvedeným podmienkam.

Súťažná úloha:

Napíš program, ktorý, keď má zadané domovské regióny všetkých zamestnancov agentúry, ako aj údaje o tom, kto je nadriadený komu, interaktívne odpovedá na otázky horeuvedeným spôsobom.

Ohraničenia:

$1 \leq N \leq 200\,000$	počet zamestnancov
$1 \leq R \leq 25\,000$	počet regiónov
$1 \leq Q \leq 200\,000$	počet otázok, na ktoré má program odpovedať
$1 \leq H_k \leq R$	domovský región zamestnanca k (pre $1 \leq k \leq N$)
$1 \leq S_k < K$	nadriadený zamestnanca k (pre $2 \leq k \leq N$)
$1 \leq r_1, r_2 \leq R$	regióny, ktorých sa týka otázka

Vstup:

Tvoj program musí načítať nasledujúce údaje zo štandardného vstupu:

Prvý riadok obsahuje celé čísla N, R a Q v tomto poradí, oddelené jednou medzerou. Ďalších N riadkov opisuje N zamestnancov agentúry v poradí podľa veku. Riadok k v týchto N riadkoch opisuje zamestnanca s číslom k. Prvý riadok (t. j. riadok opisujúci predsedu) obsahuje jedno celé číslo: domovský región H_1 predsedu. Každý z ďalších $N - 1$ riadkov obsahuje dve celé čísla oddelené jednou medzerou: pre k-teho zamestnanca jeho nadriadeného S_k a jeho domovský región H_k.

Interakcia:

Po načítaní vstupných údajov by mal tvoj program začať striedavo čítať otázky zo štandardného vstupu a vypisovať správne odpovede na štandardný výstup. Váš program musí spracovať všetkých Q otázok postupne po jednej. Presnejšie, odpoveď na otázku vždy musí vypísať skôr ako si bude môcť prečítať ďalšiu otázku.

Každú otázku dostane tvoj program ako jeden riadok štandardného vstupu, ktorý bude obsahovať dve rôzne celé čísla oddelené jednou medzerou: dva regióny r_1 a r_2.

Ako odpoveď na každú otázku vypíšte na štandardný výstup jeden riadok obsahujúci jedno celé číslo: počet dvojíc zamestnancov UNRDA e_1 a e_2, takých, že domovský región zamestnanca e_1 je región r_1, domovský región zamestnanca e_2 je r_2 a e_1 je manažérom e_2.

Môžete predpokladať, že pre ľubovoľnú otázku, ktorú vášmu programu položíme, bude odpoveď menšia ako $1\,000\,000\,000$.

Dôležitá poznámka: Aby váš program správne komunikoval s vyhodnocova-

cím programom, musí spláchnuť (flush) štandardný výstup zakaždým okamžite po tom, ako vypíše riadok s odpoveďou. Tiež sa váš program musí vyhnúť blokovaniu počas čítania štandardného vstupu. Toto by sa mohlo stať napríklad keď použijete `scanf("%d\n")`.

Bodovanie:

Za testovacie vstupy, ktoré budú mať R nanajvýš rovné 500, sa dá získať 30 bodov. Za testovacie vstupy, v ktorých nemá žiadny región viac ako 500 zamestnancov, sa dá získať 55 bodov. Za testovacie vstupy, kde platia obidve podmienky súčasne, sa dá získať 15 bodov. Za testovacie vstupy, v ktorých platí aspoň jedna z uvedených podmienok, sa dá získať 70 bodov.

Príklad:

Vstup

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

Výstup

```
1    [flush standard output]
3    [flush standard output]
2    [flush standard output]
1    [flush standard output]
```

Obchodný cestujúci

Je to už dávno, čo obchodný cestujúci zistil, že na optimálne rozvrhnutie jeho cesty Bulharskom nik nepozná efektívny algoritmus. Preto sa preorientoval na nový obchod, premenoval sa na riečneho cestujúceho a začal podnikať na rieke Dunaj (ktorú si pre naše účely predstavme ako úsečku). Kúpil si veľmi rýchlu loď s jadrovým pohonom, ktorá ho vie prepraviť medzi ľubovoľnými dvoma mestami na rieke v nulovom čase. Avšak nepostrehol istú drobnosť v technickej špecifikácii: až po kúpe zistil, že jeho loď žerie neskutočne veľa jadrového paliva. Na každý meter plavby proti prúdu rieky musí kúpiť palivo v hodnote U (ako

Up) dolárov a na každý meter plavby po prúde musí kúpiť palivo v hodnote D (ako Down) dolárov.

Pozdĺž rieky sa bude konať N trhov, ktoré by riečny cestujúci chcel navštíviť. Každý trh sa však uskutočný len v jediný deň roka. O každom trhu X riečny cestujúci vie nasledovné údaje:

– trh sa nachádza presne vo vzdialenosti L_X metrov od prameňa rieky
– bude sa konať v deň číslo T_X (deň 0 je deň, kedy si cestujúci kúpil loď)
– pokiaľ sa tohto trhu zúčastní, zarobí práve M_X dolárov.

Cestujúci potrebuje začať a aj skončiť v mieste kde má svoj sklad. Toto miesto leží vo vzdialenosti S metrov od prameňa.

Pomôžte riečnemu cestujúcemu navrhúť postupnosť trhov, ktoré má navštíviť tak, aby maximalizoval svoj zisk a začal ale aj skončil v svojom sklade. (Môže sa stať, že optimálne je nenavštíviť žiadne trhy.) Celkový zisk riečneho cestujúceho je definovaný ako celková suma dolárov, ktoré získa na trhoch, mínus celková suma dolárov, ktoré zaplatí za cestovanie po rieke.

Nezabudnite, že ak sa trh A koná skôr ako trh B, tak ich riečny cestujúci môže navštíviť oba, ale len v tomto poradí (teda nemôže ísť najskôr na trh B a potom na trh A). Avšak ak by sa dva trhy konali v ten istý deň, môže ich navštíviť oba a to v ľubovoľnom poradí. Každý deň môže riečny cestujúci navštíviť ľubovoľne veľa trhov, ktoré sa v ten deň konajú. Cez miesto, kde sa koná trh, môže prechádzať ľubovoľne veľa krát, ale navštíviť ho a zobrať príslušný zisk môže samozrejme len raz.

Súťažná úloha:

Napíš program, ktorý z daných údajov (počet trhov, údaje o nich, poloha skladu, ceny paliva) vypočíta maximálny možný zisk, ktorý môže mať riečny cestujúci po príchode naspäť do svojho skladu.

Ohraničenia:

$1 \leq N \leq 500\,000$	Počet trhov na Dunaji
$1 \leq D \leq U \leq 10$	Cena plavby 1 meter po prúde a proti prúdu
$1 \leq S \leq 500\,001$	Poloha skladu v metroch od prameňa
$1 \leq T_k \leq 500\,000$	Číslo dňa, kedy sa trh k koná
$1 \leq L_k \leq 500\,001$	Vzdialenosť v metroch trhu k od prameňa
$1 \leq M_k \leq 4\,000$	Zárobok pri návšteve trhu k

Vstup:

Tvoj program má načítať zo vstupu nasledovné dáta:

Prvý riadok vstupu obsahuje štyri celé čísla N, U, D a S, oddelené jednou

medzerou. Nasledujúcich N riadkov popisuje N trhov, pričom k-ty riadok z týchto N riadkov obsahuje tri celé čísla T_k, L_k a M_k oddelené jednou medzerou, ktoré popisujú informácie, ktoré má riečny cestujúci o trhu k.

Môžete predpokladať, že žiadne dva trhy neležia na tom istom mieste. Rovnako žiaden trh neleží na mieste, kde sa nachádza sklad riečneho cestujúceho.

Výstup:

Na štandardný výstup vypíš jediný riadok obsahujúci jediné celé číslo – maximálny zisk, ktorý vie mať riečny cestujúci po návrate späť do svojho skladu.

Bodovanie:

Vo vstupoch v hodnote 60 bodov nebudú sa žiadne dva trhy konať v ten istý deň. Vo vstupoch v hodnote 40 bodov žiadne číslo na vstupe nepresiahne hodnotu 5 000. Vo vstupoch v hodnote 15 bodov budú splnené obe predchádzajúce podmienky. Vo vstupoch v hodnote 85 bodov bude splnená aspoň jedna z prvých dvoch podmienok.

Príklad:

Vstup	Výstup
4 5 3 100 2 80 100 20 125 130 10 75 150 5 120 110	50

V optimálnom riešení riečny cestujúci navštívi najskôr trh 1 a potom trh 3 (to sú tie na pozíciách 80 a 75). Podrobný prehľad jeho cesty je nasledovný:

- Riečny cestujúci sa prepraví 20 metrov proti prúdu Dunaja, zaplatí za túto cestu 100 dolárov, a tak má zatiaľ celkový profit -100.
- Zúčastní sa trhu číslo 1, zarobí 100, a teda má celkový zisk 0.
- Pokračuje v plavbe ďalších 5 metrov v proti prúde Dunaja, zaplatí za túto cestu 25 dolárov, a tak má zatiaľ celkový zisk -25.
- Následne sa zúčastní trhu číslo 3, zarobí 150. Tým sa mu zvýši celkový zisk na 125.
- Na záver sa preplaví 25 metrov po prúde, zaplatí 75, a tak má celkový konečný zisk 50.

Stručné návody na riešenie

V tejto časti uvádzame stručné návody na riešenie úloh IOI 2009.

Plovdivská olympiáda v informatike:

Pre každú úlohu vypočítame, koľko je za ňu bodov. Následne pre každého súťažiaceho vypočítame usporiadanú trojicu: (koľko má bodov, koľko úloh vyriešil, aké má ID). Tieto trojice usporiadame a nájdeme Georga. Presnejšie, nepotrebujeme ich ani len usporiadať, stačí nájsť Georga a následne zistiť o každom inom súťažiacom, či sa umiestnil pred Georgom.

Hrozienka:

Použijeme dynamické programovanie: pre každý obdĺžnikový výrez čokolády určený rohmi (x_1, y_1) a (x_2, y_2) spočítame hodnotu H_{x_1,y_1,x_2,y_2} hovoriacu, ako najlacnejšie vieme tento kus rozlámať. Pre výrez rozmerov 1×1 je táto hodnota rovná nule, pre ľubovoľný väčší výrez ju vieme vyjadriť rekurentne: Vyskúšame všetky možnosti ako spraviť prvý rez a vyberieme najlepšiu z nich.

Nájom robotníkov:

Každý robotník i nám akoby hovorí: ak ma najmeš, tak za jednotku kvalifikácie zaplatíš aspoň $k_i = S_i/Q_i$ dolárov. Keď už máme vybratú konkrétnu skupinu robotníkov a chceme im zaplatiť čo najmenej, pozrieme sa na ich hodnoty k_i, nájdeme najväčšiu z nich (označme ju k_{max}), a následne každému vybratému robotníkovi i zaplatíme $k_{max}Q_i$ dolárov.

Je len nanajvýš N možných hodnôt k_{max}. Keď si vyberieme konkrétnu, je ňou určené, ktorých robotníkov môžeme najať, aj to, koľko ktorému z nich budeme platiť. Môžeme ich teda vyberať pažravo. Priamočiara implementácia má časovú zložitosť $O(N^2)$, pomocou haldy sa dá zlepšiť na $O(N \log N)$.

Lukostreľba:

Toto bola najťažšia úloha IOI 2009. Presné vzorové riešenie neuvedieme, len niekoľko pozorovaní, ktoré umožňujú postupne zlepšovať časovú zložitosť riešenia.

Turnaj sa určite po nanajvýš $2N$ kolách zacyklí. Najlepší lukostrelec bude stáť na terči 1, najhorších $N-1$ na terčoch 2 až N (v nejakom nešpecifikovanom poradí) a zvyšní lukostrelci budú obiehať dokola s periódou N kôl. V riešení je potrebné každý z uvedených troch prípadov spracovať samostatne.

Otázku „na ktorom terči skončím, ak začnem na terči x“ sa dá zodpovedať v lineárnom čase od N – nie je na to potrebné simulovať celý priebeh turnaja. Najzložitejšie je to v prípade, keď sme jeden z lukostrelcov 2 až $N + 1$, ktorí

budú obiehať dokola. Vtedy sa stačí dívať na to, čo sa v ktorom kole udeje na terči číslo 1. Pri tejto vylepšenej simulácii navyše nepotrebujeme evidovať ktorý lukostrelec je kde, stačí ich mať rozdelených na lepších a horších od nás.

Garáž:

Priamočiara simulácia pomocou cyklov. Efektívnejšie riešenie (ktoré ale nebolo potrebné) sa dá dosiahnuť pomocou fronty na uloženie čakajúcich áut a napríklad intervalového stromu na zapamätanie si voľných miest.

Medveď:

Najjednoduchšie naprogramovateľné riešenie za plný počet bodov vyzerá nasledovne: Na začiatku prehľadávaním do šírky pre každé políčko zistíme, kedy ho obsadia včely. Následne binárne vyhľadávame čas, ktorý môže Macko Pú zostať pri mede. Pre konkrétny čas ľahko overíme ďalším prehľadávaním do šírky, či sa ešte stíha dostať domov.

Iné možné riešenie je spracovať včely rovnako ako v prvom riešení a potom začať od konca (mackovho domu) a postupne pre ďalšie a ďalšie políčka počítať maximálny čas, kedy tam ešte môže macko stáť. Na efektívnu implementáciu potrebujeme efektívnu prioritnú frontu, keďže políčka musíme spracúvať usporiadané podľa počítaného času.

Regióny:

Na strome zamestnancov spustíme prehľadávanie do hĺbky a prečíslujeme ich, aby sme následne pre ľubovoľnú dvojicu vrcholov vedeli v konštantnom čase povedať, či je prvý z nich predkom druhého.

Pre každý región s aspoň $\sqrt{N}$ ľuďmi spustíme jedno prehľadávanie, v ktorom naraz spočítame a zapamätáme si odpovede na všetky možné otázky zahŕňajúce tento región.

Následne spracúvame otázky. Ak nám príde otázka na dva „malé“ regióny, spracujeme ju nasledovne: Každému možnému šéfovi z r_1 zodpovedá interval čísel jeho podriadených. Utriedime začiatky a konce týchto intervalov, ako aj čísla všetkých ľudí z regiónu r_2, a v lineárnom čase od ich počtu cez tieto údaje prejdeme a spočítame hľadanú odpoveď.

Obchodný cestujúci:

Najskôr vyriešime úlohu pre prípad, kedy sú všetky trhy v navzájom rôzne dni. Budeme postupne spracúvať trhy v poradí, v akom sa udejú, a pre každý z nich spočítame, aký najväčší profit vieme mať tesne po jeho navštívení.

Keď spracúvame trh, potrebujeme vedieť, odkiaľ je najlepšie naň prísť. Exis-

tuje aj lepšie riešenie ako skúšať všetky predchádzajúce trhy. Spracovaný trh x nazveme *aktívny*, ak momentálne neexistuje žiaden spracovaný trh y taký, že riešenie „ako posledný navštívim x“ je horšie ako riešenie „ako posledný navštívim y a odtiaľ docestujem na miesto konania x“. Pre každý aktívny trh x je na rieke interval miest, pre ktoré platí: ak pribudne nový trh v tomto intervale, najvýhodnejšie je naň pricestovať z trhu x.

Aktívne trhy si vieme pamätať napr. vo vyváženom strome, prípadne po predspracovaní aj v intervalovom strome.

Alternatívne riešenie je použiť dva intervalové stromy, ktoré nám umožnia pre ľubovoľnú pozíciu zistiť, aký je optimálny predchádzajúci trh po prúde a aký proti prúdu.

Viacero trhov v jeden deň spracujeme nasledovne: Stačí uvažovať cesty, ktoré začnú v jednom trhu, zaradom navštívia niekoľko ďalších a tam skončia. Najskôr teda pre každý trh v daný deň zistíme optimálny profit, keby sme navštívili len tento trh, a následne v dvoch lineárnych prechodoch (raz po prúde a raz proti prúdu) zistíme pre každý trh optimálny profit, ktorý vieme dostať presunom cez niekoľko iných trhov v daný deň.

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty štvrtý ročník Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava, 2009
142 strán, náklad 300 výtlačkov
Neprešlo jazykovou úpravou
ISBN 978-80-8072-100-8