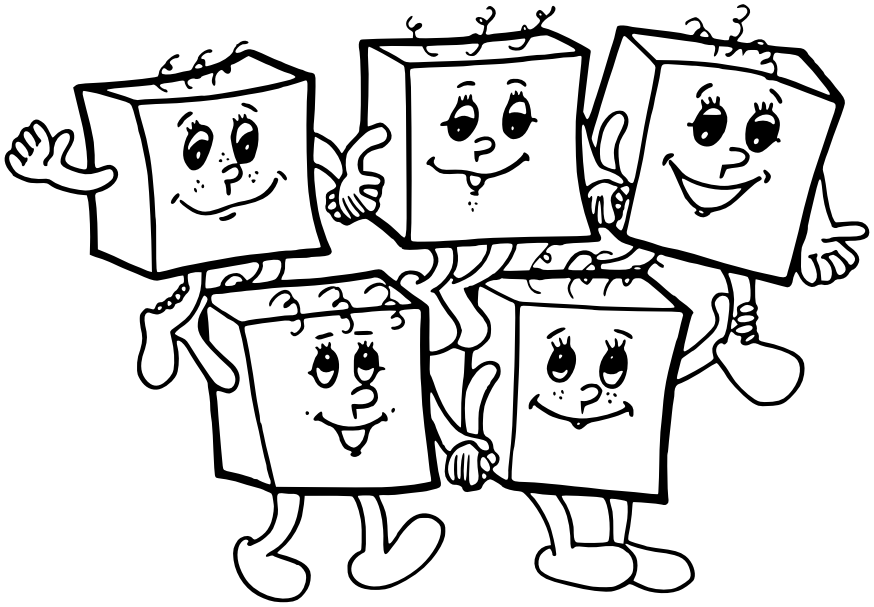
# OLYMPIÁDA V INFORMATIKE
## NA STREDNÝCH ŠKOLÁCH



**dvadsiaty štvrtý ročník**
školský rok 2008/09

**riešenia celoštátneho kola**
**kategória A**

**2. súťažný deň**

- **Olympiáda v informatike** je od školského roku 2006/07 samostatnou súťažou. Predchádzajúcich 21 ročníkov tejto súťaže prebiehalo pod názvom **Matematická olympiáda, kategória P** (programovanie).
- Oficiálnu **webstránku** súťaže nájdete na `http://oi.sk/`.

## Riešenia kategórie A

## 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.[1]

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 $N$ 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 $O(WHL)$.

[1] 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.

### 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 znamena 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");
  }
}
```

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

**DVADSIATY ŠTVRTÝ ROČNÍK OLYMPIÁDY V INFORMATIKE**

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX