A-I-1 Bežecké preteky

Úloha šla ľahko riešiť v kvadratickom čase: Pre každého bežca si spočítame jeho *poradie* v prvom aj v druhom preteku. Potom vieme pre každú dvojicu bežcov v konštantnom čase povedať, či spĺňajú podmienku zo zadania. Takéto riešenie mohlo získať až 6 alebo 7 bodov. V ďalšom texte si popíšeme, ako hľadané dvojice spočítať šikovnejšie.

Zabudnime na chvíľu na to, že bežci už sú na vstupe nejakým spôsobom očíslovaní. Namiesto toho si ich v cieli prvého preteku očíslujeme od 1 po n v poradí, v akom dobehli do cieľa. V druhom preteku si potom tieto nové čísla zapíšeme, a to opäť v poradí, v akom dobehli bežci do jeho cieľa. Pole čísel, ktoré takto dostaneme, si označme $D[1..n]$.

Dvojica bežcov, ktorí v prvom preteku dostali čísla $i < j$, spĺňa podmienku zo zadania práve vtedy, keď platí aj $D[i] < D[j]$. Spočítanie takýchto dvojíc úzko súvisí s iným známym kombinatorickým problémom: Dvojice bežcov, ktoré podmienku zo zadania *nespĺňajú*, zodpovedajú tzv. *inverziám* v poli D.

Existuje viacero rôznych algoritmov, pomocou ktorých vieme inverzie spočítať v čase $O(n \log n)$. Vieme to napríklad spraviť pomocou nejakej stromovej dátovej štruktúry (napr. intervalový strom alebo vyvažovaný binárny strom) do ktorej postupne vkladáme prvky postupnosti D a ktorá nám vie v logaritmickom čase povedať počet prvkov väčších ako ten, ktorý práve vkladáme. Asi najjednoduchší algoritmus riešiaci našu úlohu je založený na triediacom algoritme MergeSort. Priamo počas usporadúvania poľa D MergeSortom si vieme počítať dvojice, ktoré nás zaujímajú.

Celý algoritmus bude vyzerať nasledovne:

1. Rekurzívnym volaním funkcie MergeSort usporiadaj prvú polovicu poľa a spočítaj v nej dvojice, ktoré nás zaujímajú.
2. Rekurzívnym volaním funkcie MergeSort usporiadaj druhú polovicu poľa a spočítaj v nej dvojice, ktoré nás zaujímajú.
3. Spoj obe usporiadané polovice poľa do jedného usporiadaného poľa. Počas tohto spájania spočítaj také dvojice, ktorých prvý prvok je v prvej polovici a druhý v druhej.

Jedinou netriviálnou časťou je posledný krok. Ako ho spraviť šikovne? Keď spájame dve usporiadané polia, chceme pre každý prvok v prvej polovici *v konštantnom čase* zistiť počet prvkov v druhej polovici, ktoré sú od neho väčšie. Toto ale spravíme ľahko: v okamihu, keď konkrétny prvok z prvej polovice pridávame do výstupného usporiadaného poľa, stačí sa pozrieť na počet prvkov z druhej polovice poľa, ktoré sme do výstupného poľa ešte nepridali.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

long long MergeSort(vector<int> &pole) {
    // usporiada "pole" a ako návratovú hodnotu vráti počet dvojíc (i,j),
    // pre ktoré v pôvodnom poli platí i<j aj pole[i]<pole[j]
    int n = pole.size();
    if (n <= 1) return 0;
    vector<int> lave ( pole.begin(), pole.begin()+(n/2) );
    vector<int> prave( pole.begin()+(n/2), pole.end() );
    long long odpoved = 0;
    odpoved += MergeSort(lave);
    odpoved += MergeSort(prave);
    unsigned l=0, p=0;
    vector<int> pomocne;
    while (l < lave.size() && p<prave.size()) {
        if (lave[l] < prave[p]) {
            odpoved += prave.size() - p; // počet ešte nespracovaných prvkov v pravej časti
            pomocne.push_back( lave[l++] );
        } else {
            pomocne.push_back( prave[p++] );
        }
    }
    while (l < lave.size()) pomocne.push_back( lave[l++] );
    while (p < prave.size()) pomocne.push_back( prave[p++] );
    pole = pomocne;
    return odpoved;
}
```

```
int main() {
    int N; cin >> N;
    vector<int> kde(N+1);
    for (int n=0; n<N; ++n) { int x; cin >> x; kde[x] = n; }

    vector<int> D(N);
    for (int n=0; n<N; ++n) { int x; cin >> x; D[n] = kde[x]; }

    cout << MergeSort(D) << endl;
}
```

Pre zaujímavosť ešte uvádzame veľmi stručnú implementáciu v C++ využívajúcu pokročilú dátovú štruktúru: vyvažovaný binárny strom, ktorý má metódu `order_of_key(val)`. Táto metóda v logaritmickom čase vypočíta počet hodnôt ktoré sú uložené v strome a menšie ako `val`. Ide o rovnaký strom aký sa používa napr. v implementácii dátových štruktúr `set` a `map`, len si navyše v každom vrchole pamätá počet prvkov uložených v podstrome pod ním.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

typedef tree< int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update > ordered_set;

int main() {
    int N; cin >> N;
    vector<int> kde(N+1);
    for (int n=0; n<N; ++n) { int x; cin >> x; kde[x] = n; }

    vector<int> D(N);
    for (int n=0; n<N; ++n) { int x; cin >> x; D[n] = kde[x]; }

    ordered_set OS;
    long long answer = 0;
    for (int n=0; n<N; ++n) { answer += OS.order_of_key(D[n]); OS.insert(D[n]); }

    cout << answer << endl;
}
```

A-I-2 Rekonštrukcia školy

Zadanie úlohy nám hovorilo, že máme mriežku s prekážkami, v ktorej sa pohybuje štvorec 2×2. Našou úlohou bolo zistiť, či sa vieme dostať pomedzi prekážky zo začiatočnej pozície do koncovej, pričom môžeme odstrániť najviac jednu prekážku.

Ako prvé sa nám preto črtá riešenie, v ktorom pre každú prekážku vyskúšame, či neexistuje riešenie, v ktorom odstránime danú prekážku. Tým sa nám problém zjednoduší, lebo prekážka je už odstránená a my máme zistiť iba to, či sa dá dostať zo začiatku do konca nejakou voľnou cestou.

Ak by sme túto úlohu mali riešiť pre štvorec 1×1, riešenie by malo byť pomerne priamočiare. Môžeme použiť napríklad algoritmus prehľadávania do šírky, ktorý postupne zisťuje, kam všade sa vieme dostať zo začiatočného políčka. Princíp tohto algoritmu je pomerne jednoduchý – postupne objavujem všetky políčka, ktoré sú od začiatku vzdialené na 0, 1, 2 ... pohyby. Ak totiž vieme, ktoré políčka sú od začiatku vzdialené k, tak políčka vzdialené $k+1$ musia s týmito políčkami susediť a nemohli byť objavené predtým. Takto vieme o každom políčku zistiť nie len to, či sa naň dá dostať zo začiatku, ale aj to, na koľko najmenej pohybov to vieme spraviť.

Po skončení prehľadávanie iba overíme, či sa vieme dostať na políčko s koncom. Tento algoritmus je navyše pomerne rýchly, keďže každé políčko navštívi najviac raz. To vedie k zložitosti lineárnej od počtu políčok, teda $O(rs)$. Nižšie si môžete pozrieť aj implementáciu tohto algoritmu.

V našom príklade sa však pohybujeme so štvorčekom veľkosti 2×2, takže si našu mapu musíme mierne upraviť. Presnejšie, všetky pozície, na ktorých sa náš štvorec môže nachádzať sú nejaké štvorčeky 2×2. A takáto pozícia je zablokovaná a nemôžeme na ňu stúpiť, ak obsahuje aspoň jednu prekážku. Výsledná mapa bude preto o jeden riadok a jeden stĺpec kratšia. Následne už môžeme použiť vyššie spomenuté prehľadávanie do šírky.

Okrem samotného zistenia, či sa vieme dostať zo začiatku do konca však musíme aj vypísať postupnosť pohybov, ktoré máme spraviť. Ak už máme implementované prehľadávanie do šírky, vieme túto funkcionalitu naviac ľahko

pridať. Jednoducho si pre každé políčko zapamätáme, z ktorého susedného políčka sme ho objavili, poprípade ešte lepšie – z ktorého smeru sme do tohto políčka prišili. Následne vieme od konca zrekonštruovať cestu do začiatku a vypísať ju v obrátenom poradí.
Výsledná časová zložitosť takéhoto riešenia je $O(r^2s^2)$, pretože pre každú prekážku musíme spustiť jedno prehľadávanie do šírky. Takéto riešenie nám však zaručí 7 bodov a naviac sa vďaka nemu vyhneme všetkým nepríjemným špeciálnym prípadom, ktoré, ako uvidíme, nastávajú vo vzorovom riešení.

Listing programu (C++)

```
#include <cstdio>
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
#include <cmath>
#include <queue>
#include <set>
#include <map>
#include <stack>
using namespace std;

#define For(i,n) for(int i=0; i<(n); i++)
#define mp(a,b) make_pair((a),(b))
typedef long long ll;
typedef pair<int,int> pii;

int r,s;
pii zac,kon;
char mapa[2047][2047];
int spatne_hrany[2047][2047];
int dy[] = {-1,0,1,0};
int dx[] = {0,1,0,-1};
char smer[] = {'N','E','S','W'};

//najde najskorsi vyskyt znaku c
pii najdi_znak(char c) {
    For(i,r+2) For(j,s+2)
        if(mapa[i][j] == c) return mp(i,j);
}

int prekazka(char c) {
    if(c == '#') return 1;
    return 0;
}

//pocet prekazok v 2x2 stvorci
int pocet_prekazok(int y, int x) {
    int res=0;
    For(i,2) For(j,2) res += prekazka(mapa[y+i][x+j]);
    return res;
}

//algoritmus prehladavania do sirky
void prehladavanie_do_sirky(pii pozicia) {
    queue<pii> Q; //fronta v ktorej mam ulozene policka, ktore este potrebujem spracovat
    Q.push(pozicia);
    spatne_hrany[pozicia.first][pozicia.second] = -2; //oznacim si zaciatok
    while(!Q.empty()) { //kym mam co spracovavat
        pozicia = Q.front(); Q.pop(); //vyberiem prvy prvok
        For(i,4) { //pre kazdy smer
            int y=pozicia.first+dy[i], x=pozicia.second+dx[i];
            //overim ci mozem ist na danu poziciu, pripadne ci som tam uz nebol
            if(pocet_prekazok(y,x) > 0 || spatne_hrany[y][x] != -1) continue;
            spatne_hrany[y][x] = i;
            Q.push(mp(y,x)); //vlozim dalsie policko na spracovanie
        }
    }
}

void vypis_cestu(pii pozicia) {
    if(spatne_hrany[pozicia.first][pozicia.second] == -2) return;
    int p = spatne_hrany[pozicia.first][pozicia.second];
    //najskor vypisem cestu po toto policko a az potom aktualny znak
    vypis_cestu(mp(pozicia.first - dy[p], pozicia.second - dx[p]));
    printf("%c",smer[p]);
}

int main() {
    scanf("%d %d",&r,&s);
    //sentinely
    For(i,r+4) mapa[i][0]=mapa[i][s+2]=mapa[i][1]=mapa[i][s+3]='#';
    For(i,s+4) mapa[0][i]=mapa[r+2][i]=mapa[1][i]=mapa[r+3][i]='#';
    //nacitanie mapy zo vstupu
```

```
    For(i,r) For(j,s) scanf("␣%c",&mapa[i+2][j+2]);
    //najdem zaciatocnu a konecnu poziciu
    zac = najdi_znak('K');
    kon = najdi_znak('T');
    //vyskúšam odstrániť každú prekážku
    for(int y=2; y<r+2; y++)
        for(int x=2; x<s+2; x++) {
            if(mapa[y][x] != '#') continue;
            mapa[y][x] = '.';
            For(i,r+4) For(j,s+4) {
                spatne_hrany[i][j] = -1;
            }
            //prehladaj planik z policka zaciatku
            prehladavanie_do_sirky(zac);
            //ak existuje cesta do konca
            if(spatne_hrany[kon.first][kon.second] != -1) {
                vypis_cestu(kon);
                printf("\n");
                return 0;
            }
            mapa[y][x] = '#';
        }
    //vyskusam bez odstranenia prekazky
    For(i,r+4) For(j,s+4) {
        spatne_hrany[i][j] = -1;
    }
    prehladavanie_do_sirky(zac);
    if(spatne_hrany[kon.first][kon.second] != -1) {
        vypis_cestu(kon);
        printf("\n");
        return 0;
    }
    //cesta neexistuje
    printf("neexistuje\n");
}
```

Vzorové riešenie

Ak chceme naše riešenie zrýchliť, musíme si uvedomiť, čo robíme zbytočne veľakrát. Uvedomme si, že keď odstránime nejakú prekážku, ovplyvní nám to najviac 4 pozície, do ktorých táto prekážka zasahovala. Ak teda zakaždým sputíme BFS (prehľadávanie do šírky) na celom plániku, budeme veľké časti spracovávať stále dokola bez toho, aby sa v nich niečo zmenilo.

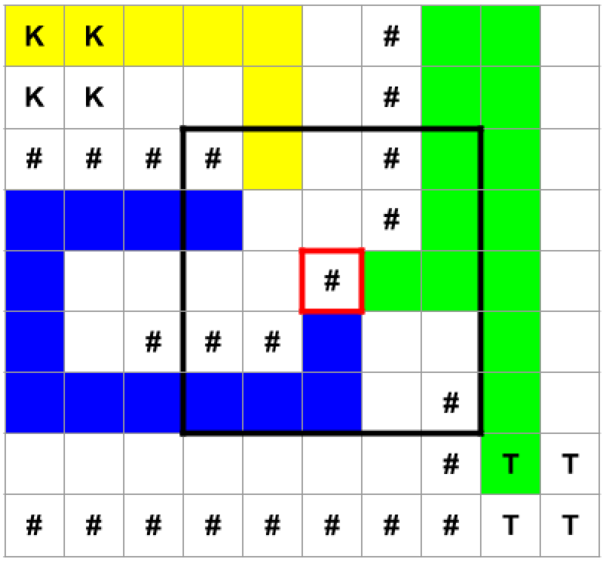
Je zjavné, že ak nejaká cesta zo začiatku do konca potrebuje odstrániť prekážku na pozícii (x, y), tak bez odstránenia ľubovoľnej prekážky (teda na pôvodnom plániku) sa táto cesta musela dostať až do stavu, ktorý tesne susedil s políčkom (x, y). To znamená, že keď uvažujeme o odstránení tejto prekážky, zaujíma nás len jej okolie veľkost 5×5 – o 2 políčka do všetkých strán. A toto je len konštantne malý výsek pôvodného plániku, na ktorom si, čo sa týka časovej zložitosti, môžem dovoliť robiť takmer všetko.

Riešenie teda môže vyzerať nasledovne: Na začiatku spustíme jedno BFS zo začiatočnej pozície, čím si vyznačíme políčka, na ktoré sa vieme dostať zo začiatku bez toho, aby sme odstránili nejakú prekážku. Ak sa medzi týmito políčkami nachádza aj cieľ, jednoducho vypíšeme cestu a môžeme skončiť.

Ak sa tam nenachádza, budeme musieť odstrániť aspoň jednu prekážku (a možno ani to nám nepomôže). Najskôr však spustíme jedno BFS aj z konca, čím zistíme, z ktorých políčok sa vieme dostať do konca. Následne sa pokúsime odstrániť každú prekážku.

Pozrieme sa na výsek 5×5 okolo tejto prekážky a túto prekážku z neho odstránime. Medzi políčkami v tomto výseku sú možno nejaké políčka, do ktorých sa vieme dostať zo začiatku a možno nejaké políčka, z ktorých sa vieme dostať do konca. Pre každú dvojicu takýchto políčok preto môžeme spustiť jedno BFS na tomto malom výseku. Ak zistíme, že medzi nimi existuje nejaká cesta, tak vieme prepojiť aj začiatok a koniec a môžeme teda vypísať odpoveď a program ukončiť.

Bohužiaľ, ako vidíme na obrázku, môže nastať aj taký prípad, v ktorom po odstránení prekážky nevieme začiatok a koniec prepojiť vnútri nášho 5×5 výseku, ale správna cesta vedie okľukou. Našťastie, počas tejto okľuky nemôžeme odstraňovať žiadnu ďalšiu prekážku a preto by sme si ju mali byť schopní predpočítať dopredu. Konkrétnejšie to bude fungovať tak, že pôvodný plánik rozdelíme na oblasti, v ktorých sa vieme pohybovať bez toho, aby sme museli odstrániť nejakú prekážku. Tieto oblasti si očíslujeme. O každom políčku teda vieme, do ktorej oblasti patrí. Nech začiatok patrí do oblasti 1 a koniec do oblasti 2. Opäť sa budeme snažiť odstrániť každú prekážku a pozerať sa iba na výsek veľkosti 5×5.



Obr. 1: Obrázok popisujúci špeciálny prípad. Žltou sú označené políčka, na ktoré sa vieme dostať zo začiatku. (Presnejšie, vieme tam mať náš ľavý horný roh.) Zelenou sú tie, z ktorých sa vieme dostať do konca. Odstraňujeme prekážku orámovanú červenou a pozeráme sa na čierny 5×5 výsek. Vidíme že po odstránení vedie cesta zo začiatku do konca po modrej ceste prekračujúcej náš výsek.

V tomto výseku však nebudeme hľadať iba priamu cestu medzi políčkom patriacim oblasti 1 a políčkom patriacim oblasti 2. Bude nám stačiť aj to, ak existuje iná oblasť k taká, že z políčka z oblasti 1 sa vieme dostať na políčko oblasti k a z políčka z oblasti 2 sa viem presunúť tiež na políčko oblasti k. Tým pádom totiž nájdeme cestu vedúcu zo začiatku do konca, ktorá prechádza oblasťou k.
Zložitosť takéhoto riešenia je už vzorových $O(rs)$, keďže na začiatku potrebujeme spustiť BFS na očíslovanie oblastí a potom už riešime všetko len na najviac rs výsekoch konštantnej veľkosti 5×5. A na takých malých výsekoch si môžeme dovoliť robiť aj pomerne veľa malých prehľadávaní.

Listing programu (C++)

```
#include <cstdio>
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
#include <cmath>
#include <queue>
#include <set>
#include <map>
#include <stack>
using namespace std;

#define For(i,n) for(int i=0; i<(n); i++)
#define mp(a,b) make_pair((a),(b))
typedef long long ll;
typedef pair<int,int> pii;

int r,s;
pii zac,kon;
char mapa[2047][2047];
int komponenty[2047][2047];
int spatne_hrany[2047][2047];
int dy[] = {-1,0,1,0};
int dx[] = {0,1,0,-1};
vector<pii> zaciatky;

//najde najskorsi vyskyt znaku c
pii najdi_znak(char c) {
    For(i,r+2) For(j,s+2)
```

```
        if(mapa[i][j] == c) return mp(i,j);
}

int prekazka(char c) {
    if(c == '#') return 1;
    return 0;
}

//pocet prekazok v 2x2 stvorci
int pocet_prekazok(int y, int x) {
    int res=0;
    For(i,2) For(j,2) res += prekazka(mapa[y+i][x+j]);
    return res;
}

//prehladame komponent, ktoremu patri policko pozicia
//takisto si vypocitame spatne hrany veduce do policka pozicia aby sme mohli
//rekonstruovat cestu
void oznac_komponent(pii pozicia, int farba) {
    queue<pii> Q; Q.push(pozicia);
    komponenty[pozicia.first][pozicia.second] = farba;
    spatne_hrany[pozicia.first][pozicia.second] = -2;
    while(!Q.empty()) {
        pozicia = Q.front(); Q.pop();
        For(i,4) {
            int y=pozicia.first+dy[i], x=pozicia.second+dx[i];
            if(komponenty[y][x] != -1 || pocet_prekazok(y,x) != 0) continue;
            komponenty[y][x] = farba;
            spatne_hrany[y][x] = i;
            Q.push(mp(y,x));
        }
    }
}

int komponent(pii kde) {
    return komponenty[kde.first][kde.second];
}

char smery[] = {'N','E','S','W'};

string otoc_smery(string s) {
    For(i,s.length()) {
        if(s[i]=='N') s[i]='S';
        else if(s[i]=='S') s[i]='N';
        else if(s[i]=='W') s[i]='E';
        else if(s[i]=='E') s[i]='W';
    }
    return s;
}

//vypis cestu z policka odkial do policka kam
string vypis_cestu(pii odkial, pii kam) {
    int komp = komponent(odkial);
    if(odkial != zaciatky[komp]) {
        string zac = vypis_cestu(zaciatky[komp], odkial);
        string kon = vypis_cestu(zaciatky[komp], kam);
        reverse(zac.begin(),zac.end());
        return otoc_smery(zac) + kon;
    }
    string res = "";
    while(spatne_hrany[kam.first][kam.second] != -2) {
        int smer = spatne_hrany[kam.first][kam.second];
        res += smery[smer];
        kam = mp(kam.first-dy[smer], kam.second-dx[smer]);
    }
    reverse(res.begin(),res.end());
    return res;
}

int nahlad_vyseku[6][6];

//pozriem sa ako vyzera okolie odstranenej prekazky
void sprav_vysek(int y, int x) {
    For(i,6) For(j,6) nahlad_vyseku[i][j] = 4;
    y-=2; x-=2;
    For(i,4) For(j,4) nahlad_vyseku[i+1][j+1] = pocet_prekazok(y+i,x+j);
}

string cesta_vo_vyseku[6][6];

//rprehladam vybrany vysek zo zadaneho policka
void prehladaj_vysek(pii odkial) {
    For(i,6) For(j,6) cesta_vo_vyseku[i][j]="!";
    queue<pii> Q; Q.push(odkial);
    cesta_vo_vyseku[odkial.first][odkial.second]="";
    while(!Q.empty()) {
        odkial = Q.front(); Q.pop();
        For(i,4) {
```

```
            int y1=odkial.first+dy[i], x1=odkial.second+dx[i];
            if(cesta_vo_vyseku[y1][x1] != "!" || nahlad_vyseku[y1][x1] != 0) continue;
            cesta_vo_vyseku[y1][x1] = cesta_vo_vyseku[odkial.first][odkial.second]+smery[i];
            Q.push(mp(y1,x1));
        }
    }
}

pii pozicia_vo_vyseku(pii povodna, pii stred) {
    return mp(povodna.first-stred.first+3,povodna.second-stred.second+3);
}

int main() {
    scanf("%d %d",&r,&s);
    //sentinely
    For(i,r+4) mapa[i][0]=mapa[i][s+2]=mapa[i][1]=mapa[i][s+3]='#';
    For(i,s+4) mapa[0][i]=mapa[r+2][i]=mapa[1][i]=mapa[r+3][i]='#';
    //nacitanie mapy zo vstupu
    For(i,r) For(j,s) scanf(" %c",&mapa[i+2][j+2]);
    //najdem zaciatocnu a konecnu poziciu
    zac = najdi_znak('K');
    kon = najdi_znak('T');
    //prehladaj vsetky suvisle komponenty, v ktorych sa vieme pohybovat bez odstranovania
    For(i,r+4) For(j,s+4) {
        komponenty[i][j] = -1;
        spatne_hrany[i][j] = -1;
    }
    For(i,r+2) For(j,s+2) {
        if(komponenty[i][j] == -1 && pocet_prekazok(i,j) == 0) {
            oznac_komponent(mp(i,j), zaciatky.size());
            zaciatky.push_back(mp(i,j));
        }
    }
    //ak su zac a kon v rovnakom komponente odpoved pozname
    if(komponent(zac) == komponent(kon)) {
        printf("%s\n",vypis_cestu(zac,kon).c_str());
        return 0;
    }
    //pokusme sa odstranit kazdu prekazku a overme, ci po jej odstraneni existuje hladana cesta
    for(int y=2; y<r+2; y++)
        for(int x=2; x<s+2; x++) {
            if(mapa[y][x] != '#') continue;
            mapa[y][x] = '.';
            //ak odstranujem prednasku, zmeni sa iba mala cast v jej okoli
            sprav_vysek(y,x);
            //zistim, ktore policka z okolia prednasky patria zaciatocnemu a ktore koncovemu komponentu
            vector<pii> zo_zaciatku,z_konca;
            for(int y1=y-2; y1<y+2; y1++)
                for(int x1=x-2; x1<x+2; x1++) {
                    if(komponent(zac) == komponent(mp(y1,x1))) zo_zaciatku.push_back(mp(y1,x1));
                    if(komponent(kon) == komponent(mp(y1,x1))) z_konca.push_back(mp(y1,x1));
                }
            //pre kazdu dvojicu policka zo zaciatocneho komponentu a z koncoveho komponentu
            //overim, ci vznikla cesta, ktora ich spaja
            For(i,zo_zaciatku.size()) For(j,z_konca.size()) {
                vector<pair<pii,string> > cesty_zo_zaciatku,cesty_z_konca;
                //do ktorych policok vyseku sa viem dostat z vybraneho zaciatocneho policka a ako vyzera cesta do nich
                prehladaj_vysek(pozicia_vo_vyseku(zo_zaciatku[i],mp(y,x)));
                For(pi,6) For(pj,6) {
                    if(cesta_vo_vyseku[pi][pj]=="!") continue;
                    cesty_zo_zaciatku.push_back(make_pair(mp(y+pi-3,x+pj-3),cesta_vo_vyseku[pi][pj]));
                }
                //do ktorych policok vyseku sa viem dostat z vybraneho koncoveho policka a ako vyzera cesta do nich
                prehladaj_vysek(pozicia_vo_vyseku(z_konca[j],mp(y,x)));
                For(pi,6) For(pj,6) reverse(cesta_vo_vyseku[pi][pj].begin(),cesta_vo_vyseku[pi][pj].end());
                For(pi,6) For(pj,6) {
                    if(cesta_vo_vyseku[pi][pj]=="!") continue;
                    cesty_z_konca.push_back(make_pair(mp(y+pi-3,x+pj-3),cesta_vo_vyseku[pi][pj]));
                }
                For(pi,cesty_zo_zaciatku.size()) For(pj,cesty_z_konca.size()) {
                    //mam dve cesty, ktore sa spajaju v tom istom bode
                    //cez tento bod teda vedie hladana cesta a uz len vypisem ako vyzera
                    if(cesty_zo_zaciatku[pi].first == cesty_z_konca[pj].first) {
                        string res = vypis_cestu(zac,zo_zaciatku[i]);
                        res += cesty_zo_zaciatku[pi].second;
                        res += otoc_smery(cesty_z_konca[pj].second);
                        res += vypis_cestu(z_konca[j],kon);
                        printf("%s\n",res.c_str());
                        return 0;
                    }
                    if(komponent(cesty_zo_zaciatku[pi].first) == -1) continue;
                    if(komponent(cesty_z_konca[pj].first) == -1) continue;
                    //z oboch vybranych policok sa viem dostat do policka patriacemu tomu istemu komponentu
                    //v tom pripade existuje cesta cez tento komponent
                    if(komponent(cesty_zo_zaciatku[pi].first) == komponent(cesty_z_konca[pj].first)) {
                        string res = vypis_cestu(zac, zo_zaciatku[i]);
                        res += cesty_zo_zaciatku[pi].second;
                        res += vypis_cestu(cesty_zo_zaciatku[pi].first,cesty_z_konca[pj].first);
```

```
                        res += otoc_smery(cesty_z_konca[pj].second);
                        res += vypis_cestu(z_konca[j],kon);;
                        printf("%s\n",res.c_str());
                        return 0;
                    }
                }
            }
            mapa[y][x] = '#';
        }
    //cesta neexistuje
    printf("neexistuje\n");
}
```

A-I-3 Moderné umenie

Riešenie tejto súťažnej úlohy úzko súvisí s tzv. *konvexným obalom* zadanej množiny bodov.
Geometrický útvar voláme *konvexný*, ak pre každé dva jeho body A, B platí, že aj celá úsečka AB je súčasťou útvaru. Neformálne, konvexný útvar „celý drží pokope“ a nenájdeme na jeho obvode žiadne „záhyby dovnútra“. *Konvexný obal* geometrického útvaru G je najmenší konvexný útvar, ktorý obsahuje G. Špeciálne platí, že konvexným obalom konečnej množiny bodov je vždy (možno degenerovaný) mnohouholník, ktorého vrcholy ležia v niektorých z dotyčných bodov. Ešte špeciálnejšie, pre body v rovine platí, že obvod ich konvexného obalu zodpovedá najkratšiemu možnému plotu, v ktorého vnútri alebo na obvode všetky body ležia.
Pre jednoznačnosť sa ešte dohodneme, že *vrcholmi* konvexného obalu našej množiny bodov budeme volať len tie body na jeho obvode, v ktorých obvod mení smer. Body ležiace na stranách konvexného obalu teda nepovažujeme za jeho vrcholy.

Malo by byť pomerne očividné, že platí nasledujúce tvrdenie: Majme danú množinu bodov v rovine, ktorú môžeme otáčať. Potom konkrétny bod z tejto množiny vieme spraviť (ostro) najpravejším práve vtedy, keď ide o jeden z vrcholov konvexného obalu.
Dôkaz: Predstavme si, že sme body ľubovoľne otočili. Ak je náš bod vo vnútri konvexného obalu, choďme teraz z neho doprava kým nenarazíme na bod, ktorý leží na jeho obvode. Tento bod je buď priamo vrcholom, ktorý leží viac napravo ako náš bod, alebo je na strane konvexného obalu, ktorej aspoň jeden koncový vrchol je viac napravo ako náš bod. Ak je náš bod na strane konvexného obalu, aspoň jeden koncový vrchol tejto strany je viac alebo tak isto napravo ako náš bod. Tým sme dokázali jednu implikáciu: AK je náš bod najviac napravo, TAK musí byť vrcholom konvexného obalu.
Opačná implikácia: Keďže ide o vrchol konvexného obalu, ten (pripomíname, že ide o mnohouholník) má pri tomto vrchole vnútorný uhol menší ako 180°. To ale znamená, že keď zoberieme os tohto uhla a natočíme rovinu tak, aby spomínaná os smerovala doľava, bude náš bod ostro napravejším zo všetkých.

Vyššie dokázané tvrdenie nám dáva pomerne priamočiare riešenie súťažnej úlohy: zostrojíme konvexný obal, overíme, či je zadaný bod jedným z jeho vrcholov, a ak áno, nájdeme os uhla pri ňom a z jej smeru vypočítame, o aký uhol treba otočiť rovinu.
Najefektívnejší algoritmus na nájdenie konvexného obalu n bodov má časovú zložitosť $O(n \log n)$. Detailný popis tohto algoritmu nájdete napr. vo vzorových riešeniach úlohy A-I-2 z 29. ročníka OI, tu ho kvôli úspore miesta vynecháme.

Naša súťažná úloha má však aj efektívnejšie riešenie, ktoré je navyše aj ľahko implementovateľné. Na to, aby sme overili, či je náš bod vrcholom konvexného obalu, nepotrebujeme celý konvexný obal zostrojiť. Stačí nám overiť, či z neho všetky ostatné body „vidíme“ pod uhlom ostro menším ako 180°.
Ako toto spraviť šikovne? Ukáže sa, že to vieme celé spraviť v lineárnom čase, na jeden prechod zoznamom bodov. Predstavme si, že sme celú rovinu posunuli tak, aby náš bod B ležal v bode $(0, 0)$. Následne si ju rozdelíme na hornú a dolnú polrovinu. Do hornej polroviny budú patriť body s kladnou y-ovou súradnicou a okrem nich ešte kladná poloos osi x, do dolnej polroviny bude patriť zvyšok.
Počas čítania bodov si budeme ku každej polrovine pamätať najmenší a najväčší z uhlov, pod ktorými vidíme body v nej. Uhol, pod ktorým vidíme bod X, meriame od kladnej poloosi osi x idúc proti smeru hodinových

ručičiek. Hornej polrovine teda zodpovedajú uhly od 0° (vrátane) po 180° (ten už nie), dolnej polrovine zase uhly od 180° (vrátane) po 360° (už nie).
Takto získaná informácia nám stačí na to, aby sme vyriešili našu súťažnú úlohu. Nás totiž zaujíma, či existuje okolo nášho bodu viac ako 180-stupňový uhol, vo vnútri ktorého žiadne body neležia. Takýto uhol môže vzniknúť len dvomi spôsobmi: buď je niektorá polrovina (horná alebo dolná) celá prázdna, alebo je dostatočný uhol medzi posledným bodom v jednej polrovine a prvým bodom v druhej.

Vyššie popísaný postup teda vedie k veľmi jednoduchému algoritmu s lineárnou časovou zložitosťou. Uhly, pod ktorými vidíme jednotlivé body, je vhodné počítať napr. funkciou `atan2(y,x)`. (Takúto funkciu máte k dispozícii vo väčšine bežných programovacích jazykov, od obyčajného arcus tangensu sa líši tým, že zohľadňuje aj kvadrant, v ktorom bod leží.)
V našom riešení sme však použili alternatívny prístup, ktorý si až do posledného kroku vystačí s celými číslami, a teda všetky výpočty spraví bez zaokrúhľovacích chýb. Namiesto explicitného výpočtu uhlov ich len porovnávame pomocou vektorového súčinu.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef complex<ll> smer;

ll cross_product(const smer &A, const smer &B) { return real(A) * imag(B) - imag(A) * real(B); }

double uhol(const smer &A) {
    double odpoved = 180. * atan2( imag(A), real(A) ) / M_PI;
    if (odpoved < 0) odpoved += 360.;
    return odpoved;
}

int main() {
    int N;
    ll bx, by;
    cin >> N >> bx >> by;

    // zostrojime si smery z nasho bodu do vsetkych inych
    vector<smer> smery;
    for (int n=0; n<N-1; ++n) {
        ll ax, ay;
        cin >> ax >> ay;
        smery.push_back( smer( ax-bx, ay-by ) );
    }

    // prejdeme vsetky smery a v kazdej polrovine si najdeme najmensi a najvacsi
    bool mam_hore = false, mam_dole = false;
    smer min_hore, max_hore, min_dole, max_dole;

    for (const auto &vec : smery) {
        bool je_hore = (imag(vec) > 0 || (imag(vec) == 0 && real(vec) > 0));
        if (je_hore) {
            if (mam_hore) {
                if (cross_product(vec,min_hore) > 0) min_hore = vec;
                if (cross_product(max_hore,vec) > 0) max_hore = vec;
            } else {
                mam_hore = true;
                min_hore = max_hore = vec;
            }
        } else {
            if (mam_dole) {
                if (cross_product(vec,min_dole) > 0) min_dole = vec;
                if (cross_product(max_dole,vec) > 0) max_dole = vec;
            } else {
                mam_dole = true;
                min_dole = max_dole = vec;
            }
        }
    }

    // zistime, ci existuje volny uhol vacsi ako 180 stupnov
    // ak existuje, vypiseme uhol, ktory otoci jeho os na kladnu poloos osi x
    if (!mam_hore) { cout << (uhol(min_dole) + uhol(max_dole))/2 - 180 << endl; return 0; }
    if (!mam_dole) { cout << (uhol(min_hore) + uhol(max_hore))/2 - 180 << endl; return 0; }
    if (cross_product(max_hore,min_dole) < 0) { cout << (uhol(min_dole) + uhol(max_hore))/2 << endl;       return 0; }
    if (cross_product(max_dole,min_hore) < 0) { cout << (uhol(max_dole) + uhol(min_hore))/2 - 180 << endl; return 0; }
    cout << "Neexistuje" << endl;
}
```

A-I-4 Stromochod

Postupne si prejdeme riešenia všetkých súťažných podúloh. Jednotlivé riešenia na sebe nezávisia, dá sa čítať každé zvlášť.

Podúloha A: overenie parity

Na overenie, či je počet vrcholov v strome párny, nám stačí jemne upraviť príklad zo študijného textu. V študijnom texte sme si ukázali program, ktorý prejde celý strom a v každom vrchole nastaví značku, že tam už bol. Náš nový program nebude všade nastavovať tú istú značku, ale na striedačku dve rôzne značky. Môžeme si to predstaviť tak, ako keby sme vrcholy, ktoré navštívime, striedavo farbili červenou a modrou. Kedy je celkový počet vrcholov párny? Práve vtedy, keď sme pri farbení posledného vrcholu použili modrú farbu.
Časová zložitosť tohto riešenia je, rovnako ako v príklade v študijnom texte, lineárna od počtu vrcholov.

Listing programu (Pascal)

```
var neparny : boolean;   { navstivil som zatial neparny pocet vrcholov? na zaciatku false }

type vrchol = record     { typ "vrchol" popisuje, ake znacky ukladame do vrcholov }
    stav: 0..3;          { pocitadlo kolkokrat sme uz tento vrchol navstivili }
    ok: boolean;         { v koreni tato znacka sluzi ako vystup, inde ju nepouzivame }
end;                     { na zaciatku je v kazdom vrchole stav = 0 }

begin
    while true do begin { nekonecny cyklus }
        V.stav := V.stav + 1;
        case V.stav of
            1: if ex_l then krok_l;
            2: if ex_p then krok_p;
            3: begin
                { naposledy opustame nejaky vrchol, zapocitame ho }
                neparny := not neparny;
                if ex_o then krok_o else begin
                    { nemame otca = sme v koreni celeho stromu, prave koncime }
                    V.ok := not neparny;
                    halt;
                end;
            end;
        end;
    end;
end.
```

Podúloha B: mocnina dvoch

To, či je nejaké číslo mocninou dvoch, vieme overiť napríklad tak, že ho dokola delíme dvojkou, až kým nedostaneme hodnotu 1.
Ako toto spraviť v našej úlohe? Spravíme niekoľko prechodov po strome a v každom z nich „zahodíme“ polovicu vrcholov. Teda presnejšie, necháme ich na mieste, len si v nich spravíme značku, že ich už ignorujeme. Ak sa tento proces zastaví s tým, že máme len jediný „nezahodený“ vrchol, bol pôvodný počet mocninou dvoch. A ak sa proces zastaví s tým, že po nejakom kole nám ostane väčší nepárny počet „nezahodených“ vrcholov, pôvodný počet vrcholov mocninou dvoch nebol.
Každý prechod n-vrcholovým stromom vieme spraviť v čase $O(n)$. Keďže v každom prechode počet nezahodených vrcholov klesne na polovicu, bude celkový počet prechodov nanajvýš $\log_2 n$, a teda celková časová zložitosť tohto algoritmu je $O(n \log n)$.

Listing programu (Pascal)

```
var neparny : boolean;   { navstivil som v aktualnom prechode neparny pocet vrcholov? }
    nechal : 0..2;       { kolko vrcholov som uz presiel a NEzahodil v aktualnom prechode?
                           hodnota 2 znamena "viac ako jeden" }

type vrchol = record     { typ "vrchol" popisuje, ake znacky ukladame do vrcholov }
    stav: 0..3;          { pocitadlo kolkokrat sme uz tento vrchol navstivili }
    ok: boolean;         { v koreni tato znacka sluzi ako vystup, inde ju nepouzivame }
    zahodeny: boolean;   { zahodili sme uz tento vrchol? na zaciatku false }
end;

procedure jeden_prechod;
```

```
begin
    while true do begin { nekonecny cyklus }
        V.stav := V.stav + 1;
        case V.stav of
            1: if ex_l then krok_l;
            2: if ex_p then krok_p;
            3: if not V.zahodeny then begin
                { sme vo vrchole, ktory sme este nezahodili, spracujeme ho }
                neparny := not neparny;
                if neparny then begin
                    { tento vrchol si nechame, zvysime pocitadlo }
                    if nechal < 2 then nechal := nechal+1;
                end else begin
                    { ale kazdy druhy vrchol zahodime }
                    V.zahodeny := true;
                end;
                { zresetujeme stav na 0 pre pripadny dalsi prechod stromu }
                V.stav := 0;
                { pohneme sa dohora, resp. skoncime tento prechod }
                if ex_o then krok_o else exit;
            end;
        end;
    end;
end;

begin
    while true do begin
        neparny := false;
        nechal := 0;
        jeden_prechod;
        if neparny then begin
            { po prechode ostal neparny pocet nezahodenych vrcholov }
            if pocet = 1 then begin
                { ostal prave jeden nezahodeny vrchol => mame mocninu dvoch }
                V.ok := true;
            end else begin
                { ostalo 3 alebo viac nezahodenych vrcholov => nemame mocninu dvoch }
                V.ok := false;
            end;
            halt;
        end;
    end;
end.
```

Podúloha C: úplný strom

To, či je strom úplný, budeme kontrolovať po úrovniach, začínajúc v koreni. Existujú len dve možnosti, ako môže vyzerať úroveň v úplnom strome: buď majú všetky vrcholy v danej úrovni dvoch synov (aj ľavého, aj pravého), alebo sme už v listoch, a teda žiaden z vrcholov nemá ani jedného syna.
Na začiatku celého programu si označíme koreň stromu. Každé kolo kontroly bude vyzerať nasledovne: Spustíme z koreňa prehľadávanie, ktoré ale nepôjde hlbšie ako po označené vrcholy. V každom označenom vrchole spravíme nasledovné:

1. odznačíme ho
2. označíme všetkých jeho synov
3. nastavíme na `true` premennú S_i, kde i je počet synov práve spracúvaného vrcholu

V premenných S_0, S_1 a S_2 si teda pamätáme, či v aktuálnej úrovni bol nejaký vrchol s 0 / 1 / 2 synmi.
Na konci prechodu sa rozhodneme nasledovne:

- Ak je pravdivé len S_0, práve spracovaná úroveň bola posledná a spracúvaný strom je naozaj úplný. V koreni o tom uložíme správu a program skončíme.
- Ak je pravdivé len S_2, práve spracovaná úroveň je korektnou vnútornou úrovňou, akú môžeme vidieť v úplnom strome. Pokračujeme na ďalší prechod, teda na kontrolu nasledujúcej úrovne.
- Vo všetkých ostatných prípadoch vieme, že náš strom nie je úplný. V koreni o tom uložíme správu a program skončíme.

Ak tento program spustíme na úplnom strome s $n = 2^k - 1$ vrcholmi, udeje sa nasledovné: pri prvom prehľadávaní sa pozrieme len na 1 vrchol, pri druhom na 1+2, pri treťom na 1+2+4, atď., až pri poslednom sa pozrieme na $1 + 2 + \cdots + 2^{k-1} =$ všetkých $2^k - 1$ vrcholov. Celkový počet navštívených vrcholov je teda menší ako $2n$.

(Iný pohľad: v poslednom prechode navštívime všetky vrcholy, v predposlednom len polovicu, o prechod skôr len štvrtinu celkového počtu, atď.)
Ak tento program spustíme na strome, ktorý úplný nie je, skončí skôr, lebo nájde spor.
V najhoršom prípade má teda tento algoritmus optimálnu časovú zložitosť $O(n)$.

Listing programu (Pascal)

```
var s0,s1,s2 : boolean;  { videl som uz v aktualnej urovni vrchol s 0/1/2 synmi? }

type vrchol = record     { typ "vrchol" popisuje, ake znacky ukladame do vrcholov }
    stav: 0..3;          { pocitadlo kolkokrat sme uz tento vrchol navstivili }
    ok: boolean;         { v koreni tato znacka sluzi ako vystup, inde ju nepouzivame }
    oznaceny: boolean;   { oznacene vrcholy tvoria prave spracuvanu uroven }
end;

procedure jeden_prechod;

begin
    while true do begin { nekonecny cyklus }
        V.stav := V.stav + 1;
        case V.stav of
            1: begin
                { prave sme prvykrat prisli do tohto vrcholu }
                if V.oznaceny then begin
                    { ako prve upravime premenne s0, s1, s2 }
                    synov := 0;
                    if ex_l and ex_p then s2 := true
                    else if ex_l or ex_p then s1 := true
                    else s0 := true;
                    { nasledne od-znacime vrchol a oznacime jeho synov }
                    V.oznaceny := false;
                    if ex_l then begin krok_l; V.oznaceny := true; krok_o; end;
                    if ex_p then begin krok_p; V.oznaceny := true; krok_o; end;
                    { a zmenime stav na 2, aby sme sa v nasledujucej iteracii uz vratili dohora }
                    V.stav := 2;
                end else begin
                    { ak aktualny vrchol nie je oznaceny, len pokracujeme dodola }
                    if ex_l then krok_l;
                end;
            2: if ex_p then krok_p;
            3: if ex_o then krok_o else exit;
            end;
        end;
    end;
end;

begin
    { na zaciatku oznacime koren }
    V.oznaceny := true;
    while true do begin
        { novy prechod, spracuvame novu uroven stromu }
        s0 := false; s1 := false; s2 := false;
        jeden_prechod;
        if s0 and (not s1) and (not s2) then begin V.ok := true; halt; end;
        if (not s0) and (not s1) and s2 then continue;
        V.ok := false;
        halt;
    end;
end.
```

TRIDSIATY DRUHÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2016