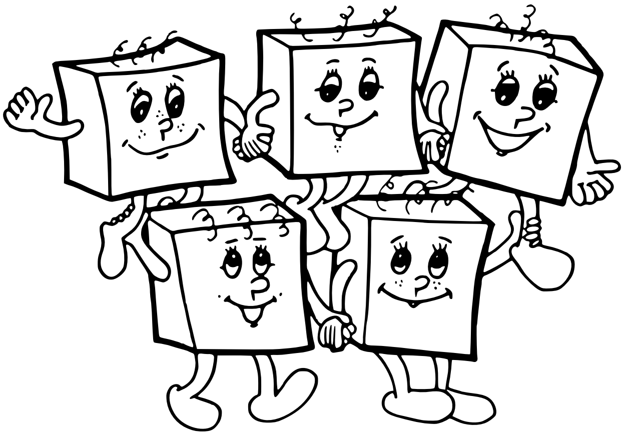
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH

http://oi.sk/



dvadsiaty šiesty ročník

školský rok 2010/11

riešenia celoštátneho kola
kategória A

2. súťažný deň

A-III-4 Asfaltistan

Zo zadanej mapy územia si najprv vytvoríme graf; jeho vrcholmi budú jednotlivé štvorcové políčka. Medzi dvoma vrcholmi bude viesť neorientovaná hrana, ak ich políčka môžeme spojiť diaľnicou, mostom alebo tunelom. Hranám priradíme ohodnotenie podľa ceny za prepojenie ich políčok.

Našou úlohou je potom nájsť najkratšiu cestu medzi vrcholmi $(0, 0)$ a $(r-1, s-1)$. Na to použijeme Dijkstrov algoritmus.

Konštrukcia grafu

Hrany reprezentujúce spojenie diaľnicou pridáme do grafu jednoducho. Stačí pre každú dvojicu susedných políčok overiť, či rozdiel ich nadmorských výšok nepresahuje 1. Keďže políčko môže mať nanajvýš štyroch susedov, časová zložitosť tejto časti aj počet pridaných hrán je $O(rs)$.

S ostatnými typmi hrán je to trochu zložitejšie. Zamerajme sa na hľadanie mostov, ktoré spájajú dve políčka v rovnakom riadku (pre vertikálny smer aj pre tunely použijeme analogický postup).

Ak chceme pre políčko (i, j) nájsť most, ktorý má na ňom svoj pravý koniec (uvedomte si, že taký môže byť najviac jeden), môžeme postupne prechádzať od neho smerom doľava, kým je nadmorská výška políčok menšia ako $h_{i,j}$. Ak narazíme na políčko s rovnakou nadmorskou výškou, našli sme most. Tento postup ale vyžaduje $O(s)$ času pre každé políčko, dokopy teda $O((r + s)rs)$.

Namiesto toho budeme spracovávať každý riadok zľava doprava (aktuálne políčko si označme (i, j)). Počas toho si budeme udržiavať zoznam tých políčok, ktoré sú z aktuálnej pozície „viditeľné“ – teda ktoré by mohla trafiť vodorovne letiaca vrana, začínajúca na aktuálnom políčku.

Napríklad ak už spracované výšky boli 100, 70, 10, 20, 20 a 14, tak viditeľné sú políčka v stĺpcoch 0, 1, 4 a 5 (výšky 100, 70, 20 a 14).

Formálne, viditeľné sú tie políčka (k, j), že $k < i$ a všetky políčka medzi (k, j) a aktuálnym majú nadmorskú výšku menšiu ako $h_{k,j}$. Všimnite si, že ak si budeme políčka v tomto zozname pamätať v poradí zľava doprava, budú ich výšky tvoriť klesajúcu postupnosť. Tiež si všimnime, že akonáhle políčko nie je viditeľné, nemôže už nikdy byť začiatkom mostu – to, že nie je viditeľné, totiž znamená, že ho zakrylo iné, aspoň rovnako vysoké políčko.

Pri spracovaní políčka (i, j) budeme postupne z konca zoznamu odstraňovať políčka, ktoré sme ním zakryli – teda všetky políčka, ktoré majú nadmorskú výšku menšiu ako $h_{i,j}$. Ak následne narazíme na políčko s výškou $h_{i,j}$, našli sme most končiaci na políčku (i, j). V opačnom prípade na aktuálnom políčku horizontálny most nekončí. Pridáme (i, j) na koniec zoznamu a ideme na nasledujúce políčko. (Keďže pridávame aj uberáme len na konci zoznamu, môžeme ho implementovať v poli ako zásobník.)

Keďže každé políčko pridáme aj odstránime zo zoznamu najviac raz, časová zložitosť spracovania jedného riadka je $O(s)$, spolu teda $O(rs)$. V tejto fáze pribudne v grafe $O(rs)$ hrán.

Dijkstrov algoritmus

V tejto časti popíšeme štandardný algoritmus na hľadanie najkratších ciest z daného začiatočného vrcholu z do všetkých ostatných vrcholov v grafe bez záporných hrán – Dijkstrov algoritmus. Označme V množinu vrcholov grafu G, množinu hrán označme E.

Počas výpočtu si budeme v poli D na pozícii i pamätať dĺžku najkratšej zatiaľ nájdenej cesty zo z do vrcholu i. Na začiatku hodnoty D inicializujeme na ∞, iba $D[z] = 0$. Navyše si budeme pre každý vrchol i pamätať, či je už hodnota $D[i]$ finálna.

V každom kroku algoritmu nájdeme taký vrchol v, ktorý má najmenšiu hodnotu $D[v]$, ale táto hodnota ešte nebola vyhlásaná za finálnu. Keďže žiadna hrana grafu nemá zápornú dĺžku, už sa nám nikdy nepodarí $D[v]$ zmenšiť, a preto je $D[v]$ finálna. Ďalej skúsime všetkým susedom i vrcholu v zlepšiť hodnotu $D[i]$ – vezmeme najkratšiu cestu z s do v, predĺžime ju do i a porovnáme jej dĺžku ($D[v] + \ell$, kde ℓ je dĺžka hrany vi) s momentálnou hodnotou $D[i]$.

Po najviac $|V|$ krokoch už budeme poznať skutočné vzdialenosti všetkých vrcholov, do ktorých sa dá zo z dostať.

Časová zložitosť algoritmu závisí od spôsobu, akým hľadáme vrchol v. Pri jednoduchom prechode všetkými vrcholmi dostaneme zložitosť $O(|V|^2)$. Ak si ale budeme hodnoty $D[i]$, ktoré ešte nie sú finálne, ukladať do minimovej haldy, najmenšiu z nich nájdeme v čase $O(\log |V|)$.

Pri zmene $D[i]$ sa však potrebujeme vysporiadať so starou hodnotou, ktorú máme v halde. Môžeme si pamätať, kde sa v halde nachádza a v prípade potreby ju odstrániť. Jednoduchším a v praxi stále dostatočne rýchlym riešením je tieto staré hodnoty v halde nechať. Nová hodnota je totiž menšia, preto ju z haldy vytiahneme skôr; keď potom niekedy vytiahneme starú hodnotu, jednoducho ju zahodíme.

Takto sa môže naraz v halde nachádzať $O(|E|)$ prvkov, čo môže byť až $O(|V|^2)$. Keďže ale $\log x^2 = 2 \log x$, časová zložitosť sa nezhoršila. Dokopy dostávame zložitosť $O(|E| \log |V|)$.

V našom konkrétnom prípade máme riedky graf ($O(rs)$ vrcholov aj hrán), preto použijeme implementáciu s haldou. Dostávame tak riešenie pôvodnej úlohy s časovou zložitosťou $O(rs \log(rs))$.

(Namiesto haldy sa dá použiť aj vyvažovaný binárny vyhľadávací strom. V ňom vieme ľahko meniť záznamy – stačí vždy zmazať starý a následne pridať nový.)

Listing programu:

```
#include <cstdio>
#include <cstdlib>
#include <vector>
#include <stack>
#include <queue>
#include <functional>
using namespace std;

struct Edge {
    int x, y;
    long long c;
    Edge(int ix, int iy, long long ic): x(ix), y(iy), c(ic) {}
};
```

```
bool operator < (const Edge &a, const Edge &b) { return a.c > b.c; }

int m, n, cD, cM, cT;
vector<vector<int> > H;
vector<vector<vector<Edge> > > G;
vector<vector<long long> > D;
priority_queue<Edge> Q;

template<typename Compare>
void vertical(int x, long long cost, Compare cmp) {
    stack<int> S;
    for (int y = 0; y < n; ++y) {
        while (!S.empty() && cmp(H[x][S.top()], H[x][y])) S.pop();
        if (!S.empty() && H[x][S.top()] == H[x][y]) {
            long long c = cD + cost * (y - S.top() - 1);
            G[x][y].push_back(Edge(x, S.top(), c));
            G[x][S.top()].push_back(Edge(x, y, c));
            S.pop();
        }
        S.push(y);
    }
}

template<typename Compare>
void horizontal(int y, long long cost, Compare cmp) {
    stack<int> S;
    for (int x = 0; x < m; ++x) {
        while (!S.empty() && cmp(H[S.top()][y], H[x][y]))
            S.pop();
        if (!S.empty() && H[S.top()][y] == H[x][y]) {
            long long c = cD + cost * (x - S.top() - 1);
            G[x][y].push_back(Edge(S.top(), y, c));
            G[S.top()][y].push_back(Edge(x, y, c));
            S.pop();
        }
        S.push(x);
    }
}

int main() {
    scanf("%d%d%d%d%d", &m, &n, &cD, &cM, &cT);
    H.resize(m, vector<int>(n));
    for (int y = 0; y < n; ++y)
        for (int x = 0; x < m; ++x)
            scanf("%d", &H[x][y]);

    int dx[4] = {0, 1, 0, -1}, dy[4] = {-1, 0, 1, 0};
    G.resize(m, vector<vector<Edge> >(n));
    for (int x = 0; x < m; ++x)
        for (int y = 0; y < n; ++y)
            for (int d = 0; d < 4; ++d) {
                int nx = x + dx[d], ny = y + dy[d];
                if (0 <= nx && nx < m && 0 <= ny && ny < n && abs(H[x][y] - H[nx][ny]) <= 1)
                    G[x][y].push_back(Edge(nx, ny, cD));
            }
    for (int x = 0; x < m; ++x) { vertical(x, cM, less<int>());   vertical(x, cT, greater<int>()); }
    for (int y = 0; y < n; ++y) { horizontal(y, cM, less<int>()); horizontal(y, cT, greater<int>());
}

    D.resize(m, vector<long long>(n, -1));
    D[m - 1][n - 1] = 0;
    Q.push(Edge(m - 1, n - 1, 0));
    while (!Q.empty()) {
        Edge e = Q.top(); Q.pop();
        if (D[e.x][e.y] < e.c)
            continue;
        for (vector<Edge>::iterator i = G[e.x][e.y].begin(); i != G[e.x][e.y].end(); ++i)
            if (D[i->x][i->y] == -1 || D[i->x][i->y] > e.c + i->c) {
                D[i->x][i->y] = e.c + i->c;
                Q.push(Edge(i->x, i->y, e.c + i->c));
            }
    }

    if (D[0][0] == -1)
        printf("NEEXISTUJE\n");
    else {
        printf("%lld\n", D[0][0] + cD);
```

```
        int x = 0, y = 0;
        while (x != m - 1 || y != n - 1) {
            printf("%d %d\n", x, y);
            for (vector<Edge>::iterator i = G[x][y].begin(); i != G[x][y].end(); ++i)
                if (D[i->x][i->y] + i->c == D[x][y]) {
                    x = i->x;
                    y = i->y;
                    break;
                }
        }
        printf("%d %d\n", x, y);
    }
}
```

A-III-5 Romantické básničky II

Pre jednoduchosť najskôr skúsime vynechať požiadavku, aby sa rýmoval začiatok a koniec básničky. Keďže je poradie slôh pevne dané, dá sa takéto zjednodušené zadanie riešiť pomocou dynamického programovania. Môžeme totiž využiť to, že ak máme množinu všetkých slôh R_i, na ktoré by báseň mohla končiť, keby mala iba i slôh, tak sme schopný ľahko nájsť množinu všetkých slôh R_{i+1}, na ktoré môže končiť $(i+1)$-slohová báseň.

Postup je jednoduchý: zo všetkých variantov slohy $i+1$, ktoré označme ako V_{i+1}, vyberieme tie, ktoré sa rýmujú, s niektorou slohou z R_i. Množina takto vybraných slôh potom tvorí množinu R_{i+1}. Ak sa na slohy budeme pozerať ako na dvojice prirodzených čísel, môžeme tento fakt zapísať matematicky ako

$$R_{i+1} = \Big\{(a,b) \;\Big|\; (a,b) \in V_{i+1} \ \wedge\ \exists x : (x,a) \in R_i\Big\}.$$

Pomocou tejto myšlienky vieme ľahko riešiť zjednodušenú úlohu. Množinu R_1 získame priamo ako množinu všetkých variantov prvej slohy, teda $R_1 = V_1$. Báseň potom postupne predlžujeme, až dostaneme množinu R_n. Keďže cieľom úlohy je vypísať vybrané varianty, musíme si zároveň s každou slohou x v R_{i+1} pamätať jej predchodcu (vďaka ktorému variantu predchádzajúcej slohy sme x vybrali do množiny R_{i+1}). Ak je takých variantov viac, stačí si pamätať ľubovoľný z nich. Pomocou tejto dodatočnej informácie môžeme ľahko odzadu zrekonštruovať niektorú z hľadaných postupností.

(Na takéto riešenie sa môžeme dívať aj ako na prehľadávanie do šírky na priestore všetkých možných „stavov počas písania básničky“ – každý stav, teda vrchol prehľadávaného grafu, vieme popísať číslom slohy, ktorú sme poslednú spracovali, a číslom jej variantu, ktorý sme si vybrali. Rovnako dobre sa dalo použiť prehľadávanie do hĺbky, je však o niečo menej názorné.)

Pridáme cyklickosť rýmov

Požiadavka na cyklickosť rýmov nám situáciu mierne skomplikuje. Nestačí totiž nájsť slohu v množine R_n, ktorá sa rýmuje s niektorou prvou slohou! Napríklad ak je prvá sloha buď $(1,2)$ alebo $(2,3)$ a druhá sloha je $(3,1)$, vieme nájsť necyklickú básničku $(2,3),(3,1)$, ale nemôžeme vziať $(1,2)$ ako prvú slohu.

Môžeme si ale napríklad s každou slohou x z množiny R_i navyše pamätať, na ktoré slohy báseň musí začínať, aby mohla končiť slohou x. Inou, trochu lepšou možnosťou je spustiť výpočet zvlášť pre každý variant prvej slohy. Ak na konci niektorá sloha z R_n končí na rým, ktorým začína aktuálne skúšaná prvná sloha, tak sme našli riešenie. Oba postupy majú rovnakú časovú zložitosť. Líšia sa ale pamäťovou zložitosťou, keďže v prvom prípade si musíme ukladať až s_1-krát viac informácií pre rekonštrukciu básničky.

Výsledný algoritmus je teraz už ľahký. Pre každý variant prvej slohy vyrobíme množinu R_1, čo je jednoprvková množina, obsahujúca vybranú prvú slohu. Použijeme výššie popísaný algoritmus pre necyklickú báseň. Ak sa v množine R_n nachádza sloha, ktorá končí na rým, na ktorý začína aktuálna prvá sloha, tak sme našli riešenie, ktoré ľahko zrekonštruujeme, vypíšeme a skončíme. V opačnom prípade pokračujeme ďalším variantom prvej slohy.

Pomalá ale funkčná implementácia

Každú množinu R_i môžeme reprezentovať booleovským poľom. To bude pre každý variant i-tej slohy určovať, či sa nachádza v množine R_i alebo nie. Pre výpočet R_{i+1} si pre každý variant vo V_{i+1} zistíme, či existuje sloha v R_i, s ktorou se rýmuje. Ak áno, vložíme tento variant do R_{i+1}.

Nech má každá sloha najviac s variantov. Potom vieme množinu R_{i+1} z množiny R_i zostrojiť v čase $O(s^2)$ – stačí každý variant vo V_{i+1} priamo porovnať s každým v R_i. Toto celé budeme robiť $O(sn)$-krát: pre každý z $O(s)$ možných začiatkov potrebujeme postupne prejsť všetkých n slôh. Celé riešenie má teda časovú zložitosť $O(ns^3)$.

Ľahká lenivá efektívna implementácia

V predchádzajúcom riešení vieme zlepšiť efektívnosť generovania množiny R_{i+1} z množiny R_i.

Ľahký spôsob je použiť hešovanie. V našom programe namiesto množiny R_i zostrojíme množinu čísel veršov, ktorými môže končiť i-ta sloha. Pre každé z nich si v hešovacej tabuľke zapamätáme číslo variantu i-tej slohy, ktorým sme ho dosiahli.

Predstavme si, že sme práve spracovali i-tu slohu. Ako dlho nám bude trvať spracovať nasledujúcu? Prejdeme všetky varianty v V_{i+1}. Každý z nich spracujeme (v očakávanom prípade) v konštantnom čase, a to nasledovne: Majme variant (a, b). Pozrieme sa do hešovacej tabuľky pre i-tu slohu. Ak sa a nedalo dosiahnuť po i-tej slohe, tento variant je nám nanič. Ak sa dalo, tak si v novej hešovacej tabuľke zaznačíme pre hodnotu b číslo aktuálneho variantu.

Takto teda každú slohu spracujeme nie v čase $O(s^2)$, ale v očakávanom čase $O(s)$. Celková časová zložitosť tohto riešenia je teda $O(ns^2)$ v očakávanom prípade. (Najhorší možný prípad je $O(ns^3)$, teda rovnaký ako pri pomalom riešení. Pri použití vhodnej hešovacej funkcie je ale prudko nepravdepodobné, že takýto zlý prípad nastane.)

Prípadne vieme dosiahnuť zaručený čas $O(ns^2 \log s)$, ak by sme namiesto hešovacej tabuľky použili asociatívne pole implementované ako vyvažovaný strom. (Teda v našom programe zmenili `unordered_map` na `map`.)

Listing programu:

```
#include <cstdio>
#include <algorithm>
#include <vector>
#include <set>
#include <tr1/unordered_map>
using namespace std;
using namespace std::tr1;

int N;
vector<int> S;
vector< vector< pair<int,int> > > verse;

void load() {
    scanf("%d",&N);
    S.resize(N);
    verse.resize(N);
    for (int n=0; n<N; ++n) {
        scanf("%d",&S[n]);
        for (int i=0; i<S[n]; ++i) {
            int x,y; scanf("%d%d",&x,&y);
            verse[n].push_back( make_pair(x,y) );
        }
    }
}

bool solve(int kde) {
    vector< unordered_map<int,int> > ako;
    ako.resize(N+1);
    ako[0][kde] = -1;
    // postupne spracuvame slohy
    for (int n=0; n<N; ++n) {
        for (int i=0; i<S[n]; ++i) {
            if (ako[n].count( verse[n][i].first )) ako[n+1][ verse[n][i].second ] = i;
        }
        if (ako[n+1].empty()) return false;
    }
    // ak sme sa dostali, kam sme chceli, vypiseme poradie
    if (!ako[N].count(kde)) return false;
    vector<int> answers;
    for (int n=N; n>0; --n) {
        answers.push_back( ako[n][kde]+1 );
        kde = verse[n-1][ ako[n][kde] ].first;
```

```
    }
    for (int n=0; n<N; ++n) printf("%d\n",answers[N-1-n]);
    return true;
}

int main() {
    load();
    set<int> candidates;
    for (int i=0; i<S[0]; ++i) candidates.insert( verse[0][i].first );
    for (set<int>::iterator it = candidates.begin(); it != candidates.end() ; ++it)
        if (solve(*it)) return 0;
    printf("NEEXISTUJE\n");
}
```

Zaručene efektívna implementácia

Upravíme spracovanie slohy na $O(s)$ v najhoršom prípade. To urobíme tak, že si v rámci predvýpočtu zoskupíme slohy z V_i do skupín tak, aby v každej boli iba slohy končiace na rovnaký rým. Teraz každú slohu x z V_{i+1} prepojíme so skupinou, ktorá zodpovedá rýmu, na ktorý x začína, prípadne si zapamätáme, že taká skupina neexistuje. Teraz môžeme jedným prechodom cez V_i pre každú skupinu zistiť, či sa v nej nachádza aspoň jedna sloha, ktorá patrí do R_i. Druhým prechodom cez V_{i+1} potom ľahko zistíme, ktoré slohy patria do R_{i+1} a ktoré nie tým, že se pozrieme na výsledok odpovedajúci skupine z predchádzajúceho prechodu.

Predvýpočet prevedieme tak, že si slohy z V_i usporiadame podľa druhého rýmu a slohy z V_{i+1} podľa prvého rýmu. Potom lineárnym prechodom očíslujeme skupiny číslami 0, 1, …, počet skupín a pre každú množinu V_i si napr. v obyčajnom poli zapamätáme, do ktorej skupiny jednotlivé varianty patria. Druhým lineárnym prechodom potom pre slohy z V_{i+1} určíme, s ktorou skupinou sú prepojené.

Časová zložitosť sa zmení na $O(ns \log s)$ na predspracovanie vstupu a $O(ns^2)$ pre samotný výpočet.

Druhé urýchlenie (ktoré ale pre dosiahnutie plného počtu bodov nebolo nutné implementovať) spočíva v myšlienke, že v skutočnosti nezáleží na tom, ktorou slohou začíname. Môžeme teda začať hľadať báseň od slohy, ktorá má najmenej variantov.

Listing programu:

```
#include <cstdio>
#include <vector>
#include <algorithm>

struct Sloka {
    Sloka() {}
    Sloka(int prvni, int druhy, int poradi)
        : PrvniRym(prvni), DruhyRym(druhy), Poradi(poradi),
          Skupina(-1), Propojeni(-1), Predchozi(-1) {}
    int PrvniRym, DruhyRym;  // cislo prvniho a druheho rymu sloky
    short int Poradi;        // poradi sloky mezi variantami
    short int Skupina;       // skupina, do ktere sloka patri
    short int Propojeni;     // skupina, s niz je sloka spojena; -1 pokud s zadnou
    short int Predchozi;     // predchozi sloka (pro rekonstrukci reseni)
};

// porovnani podle prvniho a podle druheho rymu
bool PodlePrvniho(const Sloka &leva, const Sloka &prava) { return leva.PrvniRym < prava.PrvniRym; }
bool PodleDruheho(const Sloka &leva, const Sloka &prava) { return leva.DruhyRym < prava.DruhyRym; }

int main() {
    int N;
    scanf("%d", &N);
    std::vector<std::vector<Sloka> > V(N); // seznam vsech variant

    // nacteni variant
    int maxS = 0, min = 0;
    for (int i=0; i<N; i++) {
        int S;
        scanf("%d", &S);
        for (int j=0; j<S; j++) {
            int a, b;
            scanf("%d%d", &a, &b);
            V[i].push_back(Sloka(a, b, j));
        }
        // prubezne hledane cislo sloky s nejmensim poctem variant
        if (V[i].size() < V[min].size()) min = i;
```

```
        // a take maximalni velikost sloky
        if (S > maxS) maxS = S;
    }

    // otocime vstup tak, abychom zacinali slokou s nejmensim poctem variant
    std::rotate(V.begin(), V.begin() + min, V.end());

    // predzpracovani
    for (int i=0; i<N-1; i++) {
        // setridime varianty
        std::sort(V[i].begin(), V[i].end(), PodleDruheho);
        std::sort(V[i+1].begin(), V[i+1].end(), PodlePrvniho);

        // nejdrive zaradime sloky z V[i] do skupin
        int rym = -1, skupina = -1;
        for (int j=0; j<V[i].size(); j++) {
            if (V[i][j].DruhyRym != rym) {
                V[i][j].Skupina = ++skupina;
                rym = V[i][j].DruhyRym;
            } else {
                V[i][j].Skupina = skupina;
            }
        }

        // nyni spojime sloky z V[i+1] s odpovidajicimi skupinami
        int k = 0, l = 0;
        while (k < V[i].size() && l < V[i+1].size()) {
            if (V[i][k].DruhyRym == V[i+1][l].PrvniRym) {
                V[i+1][l].Propojeni = V[i][k].Skupina;
                ++l;
            } else if (V[i][k].DruhyRym < V[i+1][l].PrvniRym) {
                ++k;
            } else {
                ++l;
            }
        }
    }

    std::vector<short int> skupiny(maxS, -1); // vysledky pro jednotlive skupiny
    std::vector<bool> R(maxS, false);         // mnozina R

    for (int prvni=0; prvni<V[0].size(); prvni++) {
        std::fill(R.begin(), R.begin() + V[0].size(), false);
        R[prvni] = true; // postupne zkousime jednotlive varianty prvni sloky

        for (int i=0; i<V.size()-1; i++) {
            std::fill(skupiny.begin(), skupiny.begin()+V[i].size(), -1);

            // spocitame, ktere skupiny obsahuji aspon jednu sloku,
            // ktera je v R, a rovnou si jednu z nich zapamatujeme
            for (int k=0; k<V[i].size(); k++) if (R[k]) skupiny[V[i][k].Skupina] = k;

            std::fill(R.begin(), R.begin()+V[i+1].size(), false);

            // do R dame ty sloky, ktere jsou spojeny se skupinou,
            // ktera obsahuje aspon jednu sloku z minule verze R
            for (int k=0; k<V[i+1].size(); k++) {
                if (V[i+1][k].Propojeni >= 0 && skupiny[V[i+1][k].Propojeni] >= 0) {
                    V[i+1][k].Predchozi = skupiny[V[i+1][k].Propojeni];
                    R[k] = true;
                }
            }
        }

        // nyni mame R spocitanou a zjistime, zda lze navazat na prvni sloku
        for (int i=0; i<V[N-1].size(); i++) {
            if (R[i] && V[N-1][i].DruhyRym == V[0][prvni].PrvniRym) {

                // nasli jsme reseni, nyni ho zrekonstruujeme
                std::vector<short int> vysledek(N);
                short int aktualni = i;
                for (int k = N-1; k >= 0; --k) {
                    // kvuli trideni slok se puvodni poradi mohlo zmenit
                    vysledek[k] = V[k][aktualni].Poradi;
                    aktualni = V[k][aktualni].Predchozi;
                }
```

```
                // vyslednou posloupnost musime otocit zpatky tak,
                // aby zacinala opet puvodni prvni slokou
                std::rotate(vysledek.begin(), vysledek.begin() +
                    ((vysledek.size() - min) % vysledek.size()), vysledek.end());

                for (int k = 0; k < N; k++) printf("%d\n", vysledek[k] + 1);
                return 0; // stacilo najit libovolne reseni, takze muzeme skoncit
            }
        }
    }

    printf("NEEXISTUJE\n"); // pokud se program dostal az sem, tak nenalezl reseni
    return 0;
}
```

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX