P-III-1 Kreslení

Triviální řešení

Celý seznam instrukcí si uložíme a budeme ho postupně procházet. Když narazíme na dotaz, projdeme všechny předchozí instrukce a budeme si udržovat, kolik pixelů již dopadlo na tázaný sloupec. Jakmile se souřadnice naplní, odpovíme barvou obdélníku (pokud se to nikdy nestane, vypíšeme -1). Dotazů je $\mathcal{O}(Q)$ a pro každý projdeme až $\mathcal{O}(Q)$ instrukcí, časová složitost je tedy $\mathcal{O}(Q^2)$.

Jedna barva

Pokud mají všechny obdélníky stejnou barvu, zajímá nás pouze, zda na tázaný sloupec již dopadlo dostatek pixelů. Chtěli bychom si tedy udržovat, kolik pixelů se právě nachází v jakém sloupci. Tyto počty potřebujeme umět rychle navyšovat pro intervaly sloupců.

Na to můžeme použít součtový intervalový strom.* Postavíme ho nad polem rozdílů sousedních počtů (například pro počty $[5, 2, 3]$ uložíme $[5, -3, 1]$). Výšku konkrétního sloupce pak získáme jako součet rozdílů v prefixu. A abychom sloupce na určitém intervalu navýšili, uděláme dvě úpravy v poli rozdílů: na levé hraně obdélníku prvek zvýšíme, za pravou hranou prvek snížíme.

Pro každou instrukci potřebujeme $\mathcal{O}(\log N)$ času, celkově je tedy časová složitost $\mathcal{O}(Q \log N)$.

Rychle a obecně

Efektivních řešení obecné úlohy existuje více, hlavní myšlenku však mají společnou: úlohu si zjednodušíme na otázku „odkdy je tázaná souřadnice zaplněná?“ a tento čas budeme binárně vyhledávat. Když zjistíme čas, podíváme se do seznamu instrukcí, jakou barvu měl příslušný obdélník. (Jestliže v době dotazu ještě nebyla souřadnice zaplněna, vypíšeme -1.)

Hodit se nám může intervalový strom z předchozí sekce. Potřebujeme akorát vědět, jak tento strom vypadá napříč časem. K tomu lze využít persistenci nebo dotazy zodpovídat paralelním binárním vyhledáváním. Oba postupy vedou k řešení v $\mathcal{O}(Q \log Q \log N)$. My si ovšem ukážeme ještě rychlejší řešení, které tyto pokročilé techniky nepotřebuje.

Plné řešení

Hlavním principem je takzvané zametání roviny. Instrukce si převedeme na události: levé hrany obdélníků, pravé hrany obdélníků a dotazy. Události následně

* Pokud tuto datovou strukturu neznáte, můžete se o ní dočíst například zde: *https://ksp.mff.cuni.cz/encyklopedie/intervalove-stromy/.*

seřadíme podle X-ové souřadnice (v případě shody bude mít přednost levá hrana, pak dotaz, nakonec pravá hrana).

Tento seznam událostí budeme postupně procházet. Narazíme-li na levou hranu, příslušný obdélník „aktivujeme“, při pravé hraně ho zase „deaktivujeme“. Díky tomu pro každý dotaz v seznamu vždy víme, jaké obdélníky na tázaný sloupec dopadly. Stačí nám tedy nějak vyhledat, jaký z nich pokryl Y-ovou souřadnici.

Opět nám pomůže součtový intervalový strom. Tentokrát ho však postavíme nad polem instrukcí, nikoli sloupců. Na začátku bude celý strom nulový, jelikož jsou všechny obdélníky deaktivované. Když však některý aktivujeme, změníme hodnotu této instrukce na výšku obdélníku.

V kořeni stromu se tudíž nachází výška aktuálního sloupce po seslání všech obdélníků. Potřebujeme zjistit, kdy tato výška dosáhla na tázanou souřadnici. Scházíme tedy po stromu dolů: je-li součet levého podstromu dostatečný, sejdeme do levého syna, jinak do pravého. Takto najdeme rozhodující obdélník a hledanou barvu. Odpověď si poznamenáme, a nakonec vše vypíšeme v originálním pořadí dotazů.

Setřídit události nám trvalo $\mathcal{O}(Q \log Q)$ času a zpracovat událost vždy $\mathcal{O}(\log Q)$ (projdeme jednu větev). Celková časová složitost je tudíž $\mathcal{O}(Q \log Q)$. Uložit musíme vstup, seznam událostí, strom a odpovědi – prostoru nám vystačí $\mathcal{O}(Q)$.

Program (C++):

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

struct Instrukce {
    bool je_dotaz;
    int c, x, w, h, y;
    int odpoved;

    // konstruktor pro obdélníky
    Instrukce(int _c, int _x, int _w, int _h) : c(_c), x(_x), w(_w), h(_h) {
        je_dotaz = false;
    }

    // konstruktor pro dotazy
    Instrukce(int _x, int _y) : x(_x), y(_y) {
        je_dotaz = true;
    }
};

vector<Instrukce> seznam_instrukci;

struct Udalost {
    int x;
    int index_instrukce;
    bool je_leva_hrana; // v případě obdélníku

    bool operator <(Udalost& druha) {
        Instrukce& tato = seznam_instrukci[index_instrukce];
        Instrukce& tamta = seznam_instrukci[druha.index_instrukce];
```

```
        // nejdůležitější je x-ová souřadnice
        if (x != druha.x) return x < druha.x;

        // prioritizujeme levé hrany
        if (druha.je_leva_hrana) return false;
        if (je_leva_hrana) return true;

        // prioritizujeme dotazy
        if (tamta.je_dotaz) return false;
        if (tato.je_dotaz) return true;

        return false; // pravé hrany, na pořadí nezáleží
    }
};

vector<Udalost> seznam_udalosti;

vector<int> intervalovy_strom;
int pocet_listu = 1;

void aktivuj(int index_obdelniku) {
    Instrukce& instrukce = seznam_instrukci[index_obdelniku];
    int vrchol = pocet_listu + index_obdelniku;
    while (vrchol > 0) {
        intervalovy_strom[vrchol] += instrukce.h;
        vrchol /= 2;
    }
}

void deaktivuj(int index_obdelniku) {
    Instrukce& instrukce = seznam_instrukci[index_obdelniku];
    int vrchol = pocet_listu + index_obdelniku;
    while (vrchol > 0) {
        intervalovy_strom[vrchol] -= instrukce.h;
        vrchol /= 2;
    }
}

int najdi_odpoved(int index_dotazu, int vyska, int vrchol = 1) {
    if (intervalovy_strom[vrchol] < vyska) {
        return -1; // pixel se nikdy nezaplní
    }

    // jde o list?
    if (vrchol >= pocet_listu) {
        int cas = vrchol - pocet_listu;
        if (cas > index_dotazu) return -1; // pixel se zaplní příliš pozdě
        else return seznam_instrukci[cas].c; // odpověď je barva obdélníku
    }

    int levy_podstrom = intervalovy_strom[2*vrchol];
    if (levy_podstrom >= vyska) {
        // sejdeme doleva
        return najdi_odpoved(index_dotazu, vyska, 2*vrchol);
    } else {
        // sejdeme doprava
        return najdi_odpoved(index_dotazu, vyska-levy_podstrom, 2*vrchol+1);
    }
}
```

```
int main() {
    int n, q;
    cin >> n >> q;

    while (pocet_listu < q) {
        pocet_listu *= 2;
    }

    intervalovy_strom.resize(2*pocet_listu);

    for (int i = 0; i < q; i++) {
        char typ;
        cin >> typ;
        if (typ == 'S') {
            // Seslat obdélník
            int c, x, w, h;
            cin >> c >> x >> w >> h;
            seznam_instrukci.push_back(Instrukce(c, x, w, h));
            seznam_udalosti.push_back({x, i, true}); // levá hrana
            seznam_udalosti.push_back({x+w-1, i, false}); // pravá hrana
        } else {
            // Dotaz
            int x, y;
            cin >> x >> y;
            seznam_instrukci.push_back(Instrukce(x, y));
            seznam_udalosti.push_back({x, i, false}); // dotaz
        }
    }

    sort(seznam_udalosti.begin(), seznam_udalosti.end());

    for (Udalost& udalost : seznam_udalosti) {
        Instrukce& instrukce = seznam_instrukci[udalost.index_instrukce];
        if (instrukce.je_dotaz) {
            instrukce.odpoved = najdi_odpoved(udalost.index_instrukce, instrukce.y);
        } else if (udalost.je_leva_hrana) {
            aktivuj(udalost.index_instrukce);
        } else {
            deaktivuj(udalost.index_instrukce);
        }
    }

    for (Instrukce& instrukce : seznam_instrukci) {
        if (instrukce.je_dotaz) {
            cout << instrukce.odpoved << "\n";
        }
    }
}
```

P-III-2 Telefony

Výlet si můžeme rozdělit na dvě samostatné části: cestu z domu k telefonu a cestu od telefonu do destinace. Obě části mohou Alici a Bobovi trvat různě dlouho, takže na sebe možná budou muset čekat – délka každé z částí závisí na nejpomalejším z nich. Formálně pokud Alice použije telefon T_a a Bob telefon T_b, celkem výlet zabere $\max(|1T_a|, |2T_b|) + \max(|T_aA|, |T_bB|)$ minut.

Všechny páry telefonů

Pro každý dotaz můžeme vyzkoušet všechna přiřazení telefonů Alici a Bobovi (těch je $\mathcal{O}(T^2)$), a nakonec zvolit to nejlepší.

Abychom spočítali, kolik času zabere výlet přes zvolené telefonní budky, potřebujeme znát několik vzdáleností a spočítat součet maxim dle vzorce výše. Triviální řešení by bylo v každém dotazu změřit každou z těchto vzdáleností pomocí prohledávání do šířky, což by stačilo na 2 body.

Můžeme si ale všimnout, že všechny tyto vzdálenosti jsou pouze od telefonních budek, kterých není mnoho – stačí tedy z každé spustit BFS a předpočítat si vzdálenosti do všech vrcholů. Takovýto předvýpočet stojí $\mathcal{O}(TM)$ času (graf je souvislý, takže $N \in \mathcal{O}(M)$) a umožní nám zodpovědět dotaz v $\mathcal{O}(T^2)$, což je za 5 bodů.

Seřazené telefony

K našemu předvýpočtu přidáme ještě seřazení telefonů, a to jak podle vzdálenosti od domu Alice, tak od domu Boba. Řadit můžeme v $\mathcal{O}(T \log T)$, případně prohledáním do šířky v $\mathcal{O}(N)$.

Pro každý dotaz budeme postupně zkoušet různé délky první části výletu (cesty k telefonu). Začneme uvažováním jen těch nejbližších telefonů, a postupně budeme přidávat další možnosti. Použít na to můžeme dva ukazatele: jeden na seřazené telefony Alice, druhý Boba. Význam ukazatele je, že právě uvažujeme daný prefix telefonů pro Alici/Boba. Když budeme chtít rozšířit, jak blízké telefony uvažujeme, posuneme nejprve ten z ukazatelů, který by pak ukazoval na bližší telefon.

Tento přístup nám řeší první část výletu – v každý moment víme, jak dlouho zrovna trvá cesta k telefonům. Jakou dvojici z nich ale zvolit, abychom minimalizovali druhou část (od telefonů do destinace)? Pro Alici i Boba si na to můžeme udržovat doposud nejbližší telefon k jejich destinaci.

Jediný problém pak nastává v situaci, kdy je preferovaný telefon obou z nich stejný, protože musí použít různé telefonní budky. Postačí si však udržovat i druhý nejbližší telefon a vyzkoušet obě možnosti, kdo oželí svůj první nejbližší.

Jelikož jsme pouze posouvali ukazatele jedním směrem, časová složitost je pro každý dotaz jen $\mathcal{O}(T)$. Dohromady to dělá $\mathcal{O}(QT + TM)$ času. Nejvíce prostoru zabírají vzdálenosti od telefonů a graf ze zadání, jinak si pamatujeme pouze seřazené telefony a pár pomocných proměnných na udržování minim. Prostorová složitost je tedy $\mathcal{O}(TN + M)$.

Program (C++):

```
#include <vector>
#include <queue>
#include <algorithm>
#include <iostream>

using namespace std;

int n, m, t, q;

const int DOMOV_ALICE = 1;
const int DOMOV_BOBA = 2;

vector<vector<int>> graf;
vector<int> telefony; // čísla kopců s telefony
vector<vector<int>> vzdalenosti_od_telefonu;
vector<int> serazene_telefony_od_alice; // indexy telefonů
vector<int> serazene_telefony_od_boba; // indexy telefonů

vector<int> spocti_vzdalenosti(int pocatek) {
    // prohledávání do šířky
    vector<int> vzdalenosti(n, -1);
    vzdalenosti[pocatek] = 0;
    queue<int> fronta({pocatek});

    while (!fronta.empty()) {
        int prvek = fronta.front();
        fronta.pop();

        for (int soused : graf[prvek]) {
            if (vzdalenosti[soused] != -1) continue;
            vzdalenosti[soused] = vzdalenosti[prvek]+1;
            fronta.push(soused);
        }
    }

    return vzdalenosti;
}

void predpocitej_vzdalenosti_od_telefonu() {
    for (int telefon : telefony) {
        vzdalenosti_od_telefonu.push_back(spocti_vzdalenosti(telefon));
    }
}

bool porovnej_vzdalenosti_telefonu_od_alice(int prvni, int druhy) {
    return vzdalenosti_od_telefonu[prvni][DOMOV_ALICE] <
           vzdalenosti_od_telefonu[druhy][DOMOV_ALICE];
}

bool porovnej_vzdalenosti_telefonu_od_boba(int prvni, int druhy) {
    return vzdalenosti_od_telefonu[prvni][DOMOV_BOBA] <
           vzdalenosti_od_telefonu[druhy][DOMOV_BOBA];
}

void serad_telefony() {
    serazene_telefony_od_alice.resize(t);
    serazene_telefony_od_boba.resize(t);
    for (int i = 0; i < t; i++) {
        serazene_telefony_od_alice[i] = i;
```

```
        serazene_telefony_od_boba[i] = i;
    }

    sort(serazene_telefony_od_alice.begin(), serazene_telefony_od_alice.end(),
         porovnej_vzdalenosti_telefonu_od_alice);
    sort(serazene_telefony_od_boba.begin(), serazene_telefony_od_boba.end(),
         porovnej_vzdalenosti_telefonu_od_boba);
}

int dotaz(int a, int b) {
    int odpoved = 1e9; // zatím nekonečno

    // aktuální telefony
    int index_alice = 0;
    int index_boba = 0;
    int alicin = serazene_telefony_od_alice[index_alice];
    int bobuv = serazene_telefony_od_boba[index_boba];

    // vzdálenosti od telefonů do destinace
    int alicin_nejlepsi = alicin;
    int od_alicina_nejlepsiho = vzdalenosti_od_telefonu[alicin][a];
    int od_alicina_druheho_nejlepsiho = 1e9;
    int bobuv_nejlepsi = bobuv;
    int od_bobova_nejlepsiho = vzdalenosti_od_telefonu[bobuv][b];
    int od_bobova_druheho_nejlepsiho = 1e9;

    // dokud nezkusíme všechny telefony
    while (true) {
        // aktuální telefony
        alicin = serazene_telefony_od_alice[index_alice];
        bobuv = serazene_telefony_od_boba[index_boba];

        // aktuální vzdálenost k telefonům
        int k_alicinu = vzdalenosti_od_telefonu[alicin][DOMOV_ALICE];
        int k_bobovu = vzdalenosti_od_telefonu[bobuv][DOMOV_BOBA];
        int k_telefonum = max(k_alicinu, k_bobovu);

        // nejlepší vzdálenost od telefonů
        int od_telefonu;
        if (alicin_nejlepsi != bobuv_nejlepsi) {
            // můžeme použít preferované telefony
            od_telefonu = max(od_alicina_nejlepsiho, od_bobova_nejlepsiho);
        } else {
            // musíme zkusit nějaký druhý nejlepší
            od_telefonu = max(od_alicina_nejlepsiho, od_bobova_druheho_nejlepsiho);
            od_telefonu = min(od_telefonu, max(od_alicina_druheho_nejlepsiho,
                                               od_bobova_nejlepsiho));
        }

        // uvažujme novou možnou nejlepší odpověď
        odpoved = min(odpoved, k_telefonum + od_telefonu);

        // nyní povolme další nejbližší telefon

        if (index_alice == t-1 && index_boba == t-1) {
            break; // už jsme zkusili všechny
        }

        int alicin_dalsi = -1;
        int k_alicinu_dalsimu = 1e9;
```

```
        int od_alicina_dalsiho = 1e9;
        if (index_alice != t-1) {
            alicin_dalsi = serazene_telefony_od_alice[index_alice+1];
            k_alicinu_dalsimu = vzdalenosti_od_telefonu[alicin_dalsi][DOMOV_ALICE];
            od_alicina_dalsiho = vzdalenosti_od_telefonu[alicin_dalsi][a];
        }

        int bobuv_dalsi = -1;
        int k_bobovu_dalsimu = 1e9;
        int od_bobova_dalsiho = 1e9;
        if (index_boba != t-1) {
            bobuv_dalsi = serazene_telefony_od_boba[index_boba+1];
            k_bobovu_dalsimu = vzdalenosti_od_telefonu[bobuv_dalsi][DOMOV_BOBA];
            od_bobova_dalsiho = vzdalenosti_od_telefonu[bobuv_dalsi][b];
        }

        if (k_alicinu_dalsimu <= k_bobovu_dalsimu) {
            // započítejme nový Alicin telefon
            index_alice++;
            if (od_alicina_dalsiho < od_alicina_nejlepsiho) {
                od_alicina_druheho_nejlepsiho = od_alicina_nejlepsiho;
                od_alicina_nejlepsiho = od_alicina_dalsiho;
            } else if (od_alicina_dalsiho < od_alicina_druheho_nejlepsiho) {
                od_alicina_druheho_nejlepsiho = od_alicina_dalsiho;
            }
        } else {
            // započítejme nový Bobův telefon
            index_boba++;
            if (od_bobova_dalsiho < od_bobova_nejlepsiho) {
                od_bobova_druheho_nejlepsiho = od_bobova_nejlepsiho;
                od_bobova_nejlepsiho = od_bobova_dalsiho;
            } else if (od_bobova_dalsiho < od_bobova_druheho_nejlepsiho) {
                od_bobova_druheho_nejlepsiho = od_bobova_dalsiho;
            }
        }
    }

    return odpoved;
}

int main() {
    cin >> n >> m >> t >> q;

    telefony.resize(t);
    for (int i = 0; i < t; i++)
        cin >> telefony[i];

    graf.resize(n);
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        u--; v--; // indexujeme od nuly
        graf[u].push_back(v);
        graf[v].push_back(u);
    }

    predpocitej_vzdalenosti_od_telefonu();
    serad_telefony();
```

```
    for (int i = 0; i < q; i++) {
        int a, b;
        cin >> a >> b;
        a--; b--; // indexujeme od nuly
        cout << dotaz(a, b) << "\n";
    }

    return 0;
}
```

P-III-3 Těžba na asteroidech

Pouze vpřed

Nejdřív vyřešme variantu pro $s = 1$. Všimněme si, že v této variantě nám stačí chodit pouze doprava. Pokud bychom chodili doleva, tak jsme na daném poli již byli a mohli těžit rovnou.

Nyní si udělejme krátký předvýpočet. Spočtěme si hodnoty $G_{s,t'}$. Ty nám udávají, kolik surovin vytěžíme, když ve sloupci s budeme těžit t' sekund:

$$G_{s,t'} = \sum_{i<t'} g_{i,s}$$

Toto jsou jen prefixové součty, takže je zvládneme spočíst v lineárním čase.

Všimněme si, že pokud skončíme ve sloupci ℓ, tak kvůli pohybu jen doprava už moc nezáleží jak jsme náš čas věnovali mezi sloupci, ale pouze maximum, které jsme schopní vytěžit. Toto napovídá na dynamické programování. – Označme si $dp[\ell][t']$ počet surovin, které stihneme vytěžit v prvních t' sekundách, když skončíme ve sloupci ℓ. Pro první sloupec se vůbec neposouváme:

$$dp[1][t'] = \begin{cases} 0 & t' = 0 \\ G_{1,t'} & \text{jinak} \end{cases}$$

Uvažme sloupec ℓ. Vyzkoušejme všechny hodnoty t_1, jak dlouho budeme chtít těžit v aktuálním sloupci. Potom jednu sekundu musíme věnovat na přesun z $\ell - 1$ na ℓ a ve zbylém času můžeme těžit. A zajímá nás optimum, tedy vezmeme maximum přes všechna t_1:

$$dp[\ell][t'] = \max_{0 \le t_1 \le t'-1} dp[\ell-1][t'-t_1-1] + G_{\ell,t_1}$$

Nyní dynamické programování stačí spočítat pro všechna $1 \le \ell \le n$ a $0 \le t' \le t$. Skončit můžeme kdekoliv, takže výsledek je $\max_{1 \le \ell \le n}$.

Máme $n \times t$ hodnot dynamického programování a výpočet jedné trvá $\mathcal{O}(t)$. Celkem tedy $\mathcal{O}(nt^2)$.

Tam i zpět

Nyní se zbavme počáteční podmínky $s = 1$. Uvažme optimální řešení, ve které vrták projde sloupci od ℓ do r. Rozdělme jeho cestu na dvě části:

1. Než se dostane do krajního sloupce ℓ nebo r.
2. Pohyb z jednoho krajního sloupce do druhého.

Část 2 nutně projde všechna políčka, takže je nejlepší jít přímo a tedy neměnit směr jinde než mezi částmi.

Z toho máme, že cesta vrtáku může vypadat jedním ze čtyř možností:

1. Jen doleva.
2. Jen doprava.
3. Doleva, pak doprava.
4. Doprava, pak doleva.

Navíc možnosti 3 a 4 musí navštívit jedno ještě nenavštívený sloupec (krajní určitě navštívený zatím nebyl). Tedy nutně musí projít znovu přes startovní sloupec s.

Pojďme tedy upravit naše dynamické programování, tak aby všechny tyto možnosti zvládlo. Možnost 1 už jsme vyřešili a možnost 2 je jen zrcadlením možnosti jedna. Zbývající možnosti znovu rozdělíme na dvě části (tentokrát jiné než předtím):

– Vracecí, kdy se vrátíme na počátek.
– Přímá, kdy z počátku jdeme jen jedním směrem.

Přímou část označme dp_p a je stejná jako za podmínky $s = 1$ (část doleva spočítáme symetricky):

$$dp_p[\ell][t'] = \max_{0 \le t_1 \le t'-1} dp_p[\ell-1][t'-t_1-1] + G_{\ell,t_1} \quad \text{pro } \ell > s$$

A vracecí část je obdobná, když si všimneme, že všechny suroviny můžeme těžit na cestě tam:

$$dp_v[\ell][t'] = \max_{0 \le t_1 \le t'-2} dp_v[\ell-1][t'-t_1-2] + G_{\ell,t_1} \quad \text{pro } \ell > s$$

Na závěr stačí vyzkoušet všechny kombinace časů strávených v obou částech:

$$\max_{0 \le t_1 \le t'} \left(\max_{1 \le \ell < s} (dp_v[\ell][t_1]) + \max_{s < r \le n} (dp_p[r][t'-t_1]) \right)$$

(A opět symetricky pro nejdřív doprava a pak doleva.)

Nyní si jen ověřme, že jsme časovou složitost nezhoršili. Hodnot dynamického programování je dvakrát tolik a počítáme je stejně rychle. Samotné počítání výsledku trvá v čase $\mathcal{O}(nt)$. Tedy jsme zase celkem na $\mathcal{O}(nt^2)$.

Program (C++):

```
#include <iostream>
#include <vector>

using std::cin;
using std::cout;
using std::vector;
using std::max;

const int INF = 1'000'000'000;

int main() {
    int n, t, s;
    cin >> n >> t >> s;
    s--;

    vector<vector<int>> g(t, vector<int>(n)), G(n, vector<int>(t+1, 0));
    for (int t0=0; t0<t; t0++) {
        for (int l=0; l<n; l++) {
            cin >> g[t0][l];
            // Spočteme prefixové součty
            G[l][t0+1] = g[t0][l] + G[l][t0];
        }
    }

    vector<vector<int>> dp_v(n, vector<int>(t+1, -INF));
    vector<vector<int>> dp_p(n, vector<int>(t+1, -INF));
    for (int t0=0; t0<t; t0++)
        dp_v[s][t0] = dp_p[s][t0] = G[s][t0];

    // Spočteme dp doleva
    for (int l=s-1; l>=0; l--) {
        for (int t0=1; t0<=t; t0++) {
            for (int t1=0; t1<=t0-1; t1++) {
                dp_p[l][t0] = max(dp_p[l][t0], dp_p[l+1][t0-t1-1] + G[l][t1]);
                if (t0 - t1 >= 2)
                    dp_v[l][t0] = max(dp_v[l][t0], dp_v[l+1][t0-t1-2] + G[l][t1]);
            }
        }
    }
    // Spočteme dp doprava
    for (int l=s+1; l<n; l++) {
        for (int t0=1; t0<=t; t0++) {
            for (int t1=0; t1<=t0-1; t1++) {
                dp_p[l][t0] = max(dp_p[l][t0], dp_p[l-1][t0-t1-1] + G[l][t1]);
                if (t0 - t1 >= 2)
                    dp_v[l][t0] = max(dp_v[l][t0], dp_v[l-1][t0-t1-2] + G[l][t1]);
            }
        }
    }

    int res = 0;

    // Jen doleva / jen doprava
    for (int l=0; l<n; l++)
        res = max(res, dp_p[l][t]);

    // S jednou otočkou
    for (int t1=0; t1<=t; t1++) {
```

```
        int lmax_v=0, lmax_p=0, rmax_v=0, rmax_p=0;
        for (int l=0; l<n; l++) {
            if (l < s) {
                lmax_v = max(lmax_v, dp_v[l][t1]);
                lmax_p = max(lmax_p, dp_p[l][t1]);
            } else if (l > s) {
                rmax_v = max(rmax_v, dp_v[l][t-t1]);
                rmax_p = max(rmax_p, dp_p[l][t-t1]);
            }
        }
        res = max(res, max(lmax_v + rmax_p, lmax_p + rmax_v));
    }

    cout << res << std::endl;
}
```