A-III-1 Tetris pre začiatočníkov

Intervalový strom pre doplnenia

Ak sme počas testu doplnili raz modré a neskôr raz červené kocky, tak v každom stĺpci, ktorý má aj modré aj červené, sú modré pod červenými. Vo všeobecnosti teda platí, že ak vieme, *ktoré* bloky kociek sú v konkrétnom stĺpci, tak je tým jednoznačne určené, *v akom sú poradí*.
Doplnenie budeme volať *aktívne*, ak zasahuje do stĺpca, na ktorý sa práve pozeráme, a *pasívne*, ak nie.
Predstavme si pole dĺžky $\leq q$, ktorého políčka zodpovedajú jednotlivým doplneniam kociek v chronologickom poradí. Na každom políčku je zapísaná buď nula, ak je zodpovedajúce doplnenie pasívne, alebo hodnota p_i, ak je toto doplnenie aktívne. Pre každé doplnenie máme teda v našom poli počet kociek, ktoré nám do práve skúmaného stĺpca pridalo. A navyše keď toto pole čítame zľava doprava, zodpovedá to tomu, ako sú kocky v našom stĺpci usporiadané zdola hore.
Nad poľom, ktoré sme si práve popísali, si postavíme klasický súčtový intervalový strom. Všimnite si, že tento intervalový strom vieme použiť na to, aby sme v stĺpci, ktorému zodpovedá, efektívne pre dané x zistili farbu x-tej kocky zdola. Toto vieme spraviť v čase $O(\log q)$ nasledovne: Stačí začať v koreni intervalového stromu a postupne zísť dodola. Kým nie sme v liste = na konkrétnom políčku poľa, vieme v každom vrchole spraviť nasledujúcu úvahu: Pozrieme sa na súčet s v našom ľavom synovi. Ten nám povie, že pridania v ňom zodpovedajú prvým s kockám. Ak je $s \geq x$, je naša hľadaná medzi nimi, pokračujeme teda do ľavého podstromu. Ak nie, pokračujeme do pravého podstromu, ale navyše si prepočítame, že v rámci neho už hľadáme nie x-tú, ale len $(x - s)$-tú kocku, keďže s sme ich už mali v ľavom podstrome.
Na začiatku budú všetky doplnenia pasívne, v celom intervalovom strome vrátane poľa samotného budú teda samé nuly.

Vzorové riešenie

Predstavme si, že prechádzame celú výslednú šachtu po stĺpcoch zľava doprava. Pre $n \leq 10^6$ by sme to naozaj mohli robiť stĺpec po stĺpci, ale pre väčšie n vieme tento prechod spraviť aj efektívnejšie. Zaujímať nás budú totiž len tie okamihy, kedy sa niečo zaujímavé stane. Sú dva typy takýchto udalostí. Za prvé, každému doplneniu zodpovedá nejaký súvislý úsek stĺpcov. Zaujímavé udalosti sú začiatok aj koniec tohto úseku, lebo vtedy toto doplnenie začne a potom znova prestane byť aktívne. No a za druhé, zaujímavé sú tie stĺpce, v ktorých sme počas testu dostali nejaké otázky.
Všetky tieto udalosti (ktorých je dokopy nanajvýš $2q$) si usporiadame do poradia, v ktorom pri prechode zľava doprava nastanú. V tomto poradí ich následne spracujeme.
Vždy, keď sa nám zmení množina aktívnych doplnení, upravíme príslušné políčko poľa a následne cestou odtiaľ do koreňa prepočítame vyššie popísaný intervalový strom. Udalosť prvého typu teda vieme spracovať v čase $O(\log q)$.
No a udalosti druhého typu? Ak máme otázku v aktuálnom stĺpci, z čísla riadku vieme povedať, koľkú kocku zdola chceme. Ak takáto kocka neexistuje (súčet v koreni celého intervalového stromu je primalý), má políčko farbu 0. V opačnom prípade vyššie popísaným algoritmom zistíme, počas ktorého doplnenia pribudla kocka na hľadanom políčku. No a na záver už len skontrolujeme, či bola naša otázka položená skôr alebo neskôr – podľa toho je odpoveď buď 0 (ešte tam kocka nebola) alebo farba kocky, ktorú sme našli.
Toto riešenie najskôr v čase $O(q)$ inicializuje prázdny intervalový strom, potom v čase $O(q \log q)$ vyrobí usporiadaný zoznam udalostí a na záver každú z $O(q)$ udalostí v čase $O(\log q)$ spracuje. Preto má celkovú časovú zložitosť $O(q \log q)$.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
const int TYP_ZAC=0, TYP_QUE=1, TYP_KON=2;
const int LOGQ = 17;

struct udalost { int s,id,typ; };
bool operator<(const udalost &A, const udalost &B) { if (A.s != B.s) return A.s < B.s; return A.typ < B.typ; }
```

```
int blok(const vector<int> &T, int x, int kde = 1) {
    if (T[kde] < x) return -1;
    if (kde >= (1 << LOGQ)) return kde - (1<<LOGQ);
    if (x <= T[2*kde]) return blok(T, x, 2*kde); else return blok(T, x-T[2*kde], 2*kde+1);
}

int main() {
    // nacitame vstup, rozdelime si ho na otazky a doplnenia, vygenerujeme udalosti
    int n, q;
    cin >> n >> q;
    vector<int> A(n), B(n), P(n), F(n), S(n), R(n), odpoved(n,-1);
    vector<udalost> U;
    for (int i=0; i<q; ++i) {
        int typ; cin >> typ;
        if (typ == 1) {
            cin >> A[i] >> B[i] >> P[i] >> F[i];
            U.push_back( {A[i], i, TYP_ZAC} );
            U.push_back( {B[i], i, TYP_KON} );
        } else {
            cin >> S[i] >> R[i];
            U.push_back( {S[i], i, TYP_QUE} );
        }
    }
    // zostrojime prazdny intervalovy strom
    vector<int> T(1<<(LOGQ+1), 0);
    // spracuvame udalosti zlava doprava
    sort( U.begin(), U.end() );
    for (auto &ud : U) {
        if (ud.typ == TYP_QUE) {
            // odpovedame na otazku
            int b = blok(T, R[ud.id]+1);
            if (b == -1 || b > ud.id) odpoved[ud.id] = 0; else odpoved[ud.id] = F[b];
        } else {
            // upravime aktivnost prave spracuvaneho intervalu
            int kde = (1 << LOGQ) + ud.id;
            if (ud.typ == TYP_ZAC) T[kde] = P[ud.id]; else T[kde] = 0;
            // prepocitame intervalovy strom
            while (true) {
                kde /= 2; if (kde == 0) break;
                T[kde] = T[2*kde] + T[2*kde+1];
            }
        }
    }
    // vypiseme odpovede
    for (int x : odpoved) if (x != -1) cout << x << endl;
}
```

Alternatívne riešenie

Pre $n \approx 10^6$ vieme vyššie popísané riešenie implementovať aj o čosi ľahšie tak, že si priamo ku každému stĺpcu šachty zapíšeme udalosti, ktoré tam nastávajú. Potom naozaj môžeme postupne ísť zľava doprava pozrieť na každý stĺpec šachty a vtedy zodpovedať všetky otázky o ňom.

No a keď už máme toto riešene, plné zadanie s $n \leq 10^9$ naň vieme previesť ľahkým trikom známym pod menom *kompresia súradníc*. Pri tej zoberieme úplne všetky čísla stĺpcov, ktoré sa vo vstupe naozaj nachádzajú. Všetky výskyty najmenšieho z nich nahradíme číslom 0, druhé najmenšie číslo nahradíme číslom 1, tretie najmenšie číslom 2, a tak ďalej.

Čo takto dostaneme? Na jednej strane vieme povedať, že vo vstupe s q operáciami je nanajvýš $2q$ rôznych čísel stĺpcov, takže po našej úprave majú všetky použité stĺpce čísla menšia ako $2q$. No a na druhej strane si ľahko odvodíme, že takáto úprava nijak nezmení odpovede na otázky. Rozmyslite si, že sme vstup len „vhodne zúžili“. (V pôvodnom vstupe mohli byť dlhé súvislé bloky rovnakých stĺpcov, v ktorých sa nič zaujímavé nedeje. Kompresia súradníc každý takýto blok zredukuje na jediný stĺpec.)

Stručne o trochu pomalších riešeniach

Predstavme si, že máme pole, v ktorom máme pre každý stĺpec šachty počet kociek v ňom. Na začiatku sú v tomto poli samé nuly. Vždy, keď dopĺňame nové kocky, potrebujeme k hodnotám v súvislom úseku nášho poľa (od a_i po b_i) pripočítať konštantu p_i. Následne ešte budeme potrebovať operáciu, kedy chceme prečítať konkrétnu hodnotu v poli, t.j., pre konkrétny stĺpec zistiť, koľko aktuálne obsahuje kociek.

Tieto operácie vieme efektívne robiť pomocou intervalového stromu. Existuje viacero spôsobov.

Jedna možnosť je priamo vkladať intervaly, pričom úpravy stromu robíme lenivo (angl. *lazy*) – vo vnútorných vrcholoch si pamätáme, o koľko treba všetko pod ním zväčšiť, a až keď cez takýto vrchol potrebujeme prejsť

dodola, prepošleme túto informáciu hlbšie.
Iná, trikovejšia možnosť, je namiesto vyššie popísaného poľa P udržiavať pole Q, v ktorom $Q[i] = P[i] - P[i-1]$, inými slovami, rozdiely medzi susednými hodnotami v poli P. Môžeme si všimnúť, že keď v poli P zväčšujeme interval hodnôt o konštantu, v poli Q sa zmenia len dve hodnoty – na začiatku a na konci zmeneného úseku. A tiež si môžeme všimnúť, že konkrétnu hodnotu poľa P vieme určiť ako súčet príslušne dlhého prefixu poľa Q. Stačí nám teda na všetky potrebné operácie postaviť nad poľom Q obyčajný súčtový intervalový strom.
Bez ohľadu na to, ktorú možnosť sme si vybrali, teraz budeme pokračovať rovnako. Aby sme vedeli odpovedať na otázku o farbe kocky na konkrétnom políčku, stačí nám vedieť zistiť, do ktorého doplnenia táto kocka patrí. Inými slovami, ak v nejakom stĺpci s hľadáme x-tú kocku zdola, tak chceme nájsť okamih v čase, kedy prvýkrát počet kociek v tomto stĺpci dosiahol aspoň x.
Ak by sme vedeli cestovať v čase, vedeli by sme hľadaný čas t binárne vyhľadať. Potrebovali by sme sa v $O(\log q)$ rôznych časoch pozrieť na počet kociek v stĺpci s. Každú takúto otázku by sme zodpovedali tak, že sa na to v príslušnom čase opýtame vtedy-aktuálneho intervalového stromu.
No a toto naozaj vieme efektívne spraviť. Intervalové stromy vieme upraviť do tzv. *perzistentnej* verzie. Takto upravené stromy si pamätajú nielen svoj súčasný stav, ale taktiež vieme zrekonštruovať, ako celý strom vyzeral v ľubovoľnom skoršom okamihu. Detaily jednej možnej konštrukcie si môžete napr. pozrieť v tomto videu: https://www.youtube.com/watch?v=bmSa2HAPtE8.

A-III-2 Telefónne búdky

Vzdialenosť vrcholov x a y (a teda aj čas v minútach potrebný na prechod medzi nimi) budeme označovať $|xy|$.

Riešenie za 5 bodov

Ako by sme úlohu vedeli vyriešiť, keby sme vedeli, že Alica použije búdku v lokalite f_a a Bob v f_b?
Každú cestu z domu do cieľa si môžeme rozdeliť na dve časti: z domu po búdku a od búdky po cieľ. Cesta k búdke bude Alici trvať $|d_a f_a|$ a Bobovi $|d_b f_b|$. Aby si mohli zavolať, musí rýchlejší z nich počkať na pomalšieho. Telefonovať sa teda začne v čase $\max(|d_a f_a|, |d_b f_b|)$. Po telefonáte sa obaja vyberú od svojich búdok do svojich cieľov a opäť bude rýchlejší z nich čakať na pomalšieho – výlet skončí, až keď sa do cieľov dostanú obaja.
Na to, aby sme spočítali optimálny čas výletu pre tieto konkrétne búdky f_a a f_b nám teda stačí poznať štyri vzdialenosti: vyššie uvedené vzdialenosti domov od búdok a tiež vzdialenosti $|f_a c_a|$ a $|f_b c_b|$ búdok od cieľov.

Teraz je kľúčové všimnúť si, že každá vzdialenosť, ktorú potrebujeme, je medzi nejakou búdkou a nejakým vrcholom. Postupne pre každú búdku zvlášť spustíme jedno prehľadávanie do šírky (BFS), ktoré začne od dotyčnej búdky a vypočíta nám v čase $O(m)$ vzdialenosti od nej do všetkých ostatných vrcholov. Takto vypočítané vzdialenosti si zapamätáme v tabuľke. Keďže búdiek máme t, dokopy strávime týmto predpočítaním $O(tm)$ času. Vypočítaná tabuľka vzdialeností medzi búdkami a vrcholmi nám zaberie $O(tn)$ pamäte.

S takto predpočítanými vzdialenosťami teraz už ľahko vyriešime našu súťažnú úlohu za 5 bodov, teda v čase $O(mt + qt^2)$. Stačí totiž pre každú z q otázok postupne vyskúšať všetkých $t(t-1)$ možností pre to, z ktorej búdky f_a bude volať Alica a z ktorej búdky f_b bude volať Bob. Pre každú dvojicu (f_a, f_b) vieme v konštantnom čase spočítať čas, ktorý by celý výlet trval, no a z týchto výletov si už stačí len vybrať najkratší.

Vylepšenie na vzorové riešenie

Predstavme si, že namiesto toho, aby sme dostali predpísané, ktoré búdky Alica a Bob použijú, budeme tentokrát mať obmedzenie, že si majú zavolať presne po k minútach výletu. Ako vyzerá najlepšie riešenie s týmto dodatočným parametrom?
Alica si môže vybrať spomedzi búdok, ku ktorým sa vie z domu dostať za k minút. Ktorú z nich sa jej oplatí vybrať? Keďže je predpísané, kedy sa telefonuje, dĺžku výletu vie ovplyvniť už len jediné: to, ako dlho to po telefonáte bude Alici trvať do cieľa. Spomedzi búdiek, ktoré vie dosiahnuť, si teda chce vybrať tú, z ktorej je to do cieľa aktuálneho výletu najbližšie. Presne rovnako môže uvažovať aj Bob: z búdiek, ktoré on vie dosiahnuť, si chce vybrať najbližšiu k jeho cieľu.

Ak sme takto vybrali každému inú búdku, máme pre aktuálne k optimálny výlet. Čo ale so situáciou, kedy je pre Alicu optimálna tá istá búdka ako pre Boba? Samozrejme, nemôžu si ju vybrať obaja. Tu je ale ľahká pomoc – stačí vyskúšať, kto ju nepoužije, a vybrať lepšiu z týchto dvoch možností. Ak sa napríklad rozhodneme, že Alica túto búdku nepoužije, je jasné, že pre Boba je stále optimálne ju použiť. A takisto si ľahko rozmyslíme, že pre Alicu je následne optimálne vybrať si najlepšiu zo zostávajúcich búdiek, ktoré vie dosiahnuť.

Aby sme vedeli efektívne zistiť, ktoré búdky sú kedy dostatočne blízko, zoraďme si všetky podľa vzdialenosti od Alicinho a tiež podľa vzdialenosti od Bobovho domu. Toto môžeme spraviť v čase $O(m)$ ďalšími dvoma prehľadávaniami do šírky. Iné, porovnateľne dobré riešenie je uvedomiť si, že všetky vzdialenosti od búdok k domom už máme v predpočítaných dátach, stačí ich povyberať a usporiadať (buď všeobecným algoritmom v čase $O(t \log t)$ alebo countsortom v čase $O(n)$).
Predstavme si teraz, že začneme s $k = 0$ (vtedy ešte nie sú dosiahnuteľné žiadne búdky a teda sa výlet nedá uskutočniť) a postupne budeme k zvyšovať. V niektorých okamihoch pribudne nová búdka dosiahnuteľná buď Alicou alebo Bobom. Optimálny výlet zjavne musí zodpovedať niektorému z týchto okamihov – nemá zmysel, aby pri búdkach obaja čakali, v optimálnom riešení sa určite telefonuje len čo pomalší z nich príde ku svojej búdke. Stačí teda postupne prejsť cez týchto nanajvýš $2t$ rôznych hodnôt k, pre každú z nich zistiť najkratší jej zodpovedajúci výlet a vypísať najlepšie spomedzi takto získaných riešení.
Ako toto vieme spraviť efektívne? Aj pre Alicu aj pre Boba si budeme pamätať dve najlepšie spomedzi nimi dosiahnuteľných búdok – teda tie dve z nich, z ktorých to je najbližšie do príslušného cieľa. Keď niekomu pribudne nová dosiahnuteľná búdka, spracujeme ju v konštantnom čase: len sa pozrieme na jej vzdialenosť od cieľa a ak je dostatočne nízka, zaradíme ju medzi pamätané dve najlepšie. Po každej takejto zmene vieme taktiež v konštantnom čase vypočítať, ako dlho bude trvať najlepší výlet zodpovedajúci aktuálnemu k.
Takéto riešenie teda každú otázku spracuje v čase $O(t)$, čím dostávame práve potrebnú časovú zložitosť $O(mt + qt)$. Celková pamäťová zložitosť je $O(m + tn)$.

Listing programu (C++)

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

int n, m, t, q, da, db;
vector<int> B;
vector< vector<int> > G;
vector< vector<int> > vzdialenost;

vector<int> bfs(int start) {
    // klasicke prehladavanie do sirky z vrcholu start
    vector<int> odpoved(n, n+47);
    odpoved[start] = 0;
    queue<int> Q;
    Q.push(start);
    while (!Q.empty()) {
        int kde = Q.front(); Q.pop();
        for (int kam : G[kde]) if (odpoved[kam] > odpoved[kde]+1) {
            odpoved[kam] = odpoved[kde]+1;
            Q.push(kam);
        }
    }
    return odpoved;
}

void pridaj_budku(int &prva, int &druha, int nova, int ciel) {
    if (prva == -1) { prva = nova; return; }
    if (vzdialenost[nova][ciel] < vzdialenost[prva][ciel]) { int x=nova; nova=prva; prva=x; }
    if (druha == -1) { druha = nova; return; }
    if (vzdialenost[nova][ciel] < vzdialenost[druha][ciel]) druha = nova;
}

int najlepsi_chvost(int na1, int na2, int ca, int nb1, int nb2, int cb) {
    int odpoved = n+7;
    for (int na : {na1, na2}) for (int nb : {nb1, nb2}) if (na != nb && na != -1 && nb != -1)
        odpoved = min( odpoved, max(vzdialenost[na][ca], vzdialenost[nb][cb]) );
    return odpoved;
}

int main() {
    // nacitame konstantnu cast vstupu
    cin >> n >> m >> t >> q >> da >> db;
    B.resize(t); for (int &b : B) cin >> b;
    G.resize(n);
    for (int i=0; i<m; ++i) { int x,y; cin >> x >> y; G[x].push_back(y); G[y].push_back(x); }
```

```
    // predpocitame vzdialenosti od budiek do sveta + poradia od domov po budky
    vector< pair<int,int> > poradieA( 1, {n+7,-1} ), poradieB( 1, {n+7, -1} ); // zarážky
    for (int i=0; i<t; ++i) {
        vzdialenost.push_back( bfs( B[i] ) );
        poradieA.push_back( { vzdialenost[i][da], i } );
        poradieB.push_back( { vzdialenost[i][db], i } );
    }
    sort( poradieA.begin(), poradieA.end() );
    sort( poradieB.begin(), poradieB.end() );

    // ideme odpovedat na otazky
    for (int qq=0; qq<q; ++qq) {
        // nacitame ciele vyletu, inicializujeme si najlepsie budky
        int ca, cb; cin >> ca >> cb;
        int na1=-1, na2=-1, nb1=-1, nb2=-1;
        int najlepsi = 2*n+7;
        // postupne pridavame budky a prepocitavame vylet
        int ia=0, ib=0, cas=0;
        for (int kolo=0; kolo<2*t; ++kolo) {
            if (poradieA[ia].first <= poradieB[ib].first) {
                pridaj_budku(na1, na2, poradieA[ia].second, ca);
                cas = poradieA[ia].first;
                ++ia;
            } else {
                pridaj_budku(nb1, nb2, poradieB[ib].second, cb);
                cas = poradieB[ib].first;
                ++ib;
            }
            najlepsi = min( najlepsi, cas+1+najlepsi_chvost(na1,na2,ca,nb1,nb2,cb) );
        }
        cout << najlepsi << endl;
    }
}
```

A-III-3 Ryžovanie

Začnime vyriešením ľahšej úlohy, v ktorej sa po rieke môžeme hýbať len jedným smerom (napr. len po prúde). Pri riešení tejto úlohy si môžeme klásť otázky nasledovného tvaru: „Koľko najviac zlata viem získať, ak začínam na nedotknutom úseku i a mám j minút času?“ Odpoveď na túto otázku si označíme $B_{i,j}$.

Označme $Z_{i,j} = z_{i,1} + \cdots + z_{i,j}$. Slovne, $Z_{i,j}$ je celkové množstvo zlata, ktoré dostaneme na úseku i počas prvých j minút ryžovania. (Všetky sčítance, ktoré neboli zadané na vstupe, sú nuly. Inými slovami, pre $j > m$ platí $Z_{i,j} = Z_{i,m}$.)

Pre $i = n$ sa už nemáme kam inam pohnúť, a teda máme len jedinú možnosť: j minút ryžovať na úseku, kde sme. Odpoveďou na našu otázku je teda hodnota $Z_{n,j}$.

Pre menšie i máme na výber, koľko minút na začiatku strávime na úseku i. Aký najväčší zisk vieme mať, ak ich bude presne k? Z úseku i dostaneme $Z_{i,k}$ gramov zlata. Potom strávime jednu minútu (ak ju ešte máme, teda ak $k < j$) presunom o úsek ďalej. No a následne sme na nedotknutom úseku $i + 1$ a máme ešte $j - k - 1$ minút času. Optimálny spôsob, ako pokračovať ďalej a získať za tento zvyšný čas čo najviac zlata, teda opäť zodpovedá jednej z našich otázok. Vieme teda povedať, že ak na úseku i strávime k minút, najväčšie množstvo zlata, ktoré vieme získať, bude presne $Z_{i,k} + B_{i+1,j-k-1}$. No a optimálnu odpoveď na našu pôvodnú otázku, teda optimálne riešenie pre konkrétne i a j, vieme nájsť tak, že vyskúšame všetky možnosti pre k a vyberieme najlepšiu z nich. Platí teda nasledovný vzťah:

$$B_{i,j} = \max_{0 \le k \le j} Z_{i,k} + B_{i+1,j-k-1}$$

Toto nám už umožní postupne vypočítať všetky hodnoty $B_{i,j}$. Začneme tým, že priamo vypočítame všetky hodnoty $B_{n,?}$, potom z nich vyššie uvedeným vzorcom vypočítame všetky hodnoty $B_{n-1,?}$, z nich všetky hodnoty $B_{n-2,?}$ a tak ďalej.

Dokopy takto spočítame $O(nt)$ rôznych hodnôt $B_{i,j}$. Každú konkrétnu hodnotu $B_{i,j}$ pre $i < n$ vieme vypočítať vyskúšaním $O(t)$ možností, dokopy teda týmto výpočtom strávime $O(nt^2)$ krokov.

Ale vieme ešte trochu ušetriť: pre $i < n$ a $j > m$ si môžeme všimnúť, že nemá zmysel skúšať možnosti s $k > m$, keďže $Z_{i,k}$ už ostáva konštantné a druhý sčítanec sa nezväčšuje (za menej času nevieme v rovnakej situácii vyryžovať viac zlata). Preto každú konkrétnu hodnotu vieme určiť v čase $O(m)$, čo nám zlepší odhad časovej zložitosti na $O(nmt)$.

Rovnaký výpočet vieme spraviť aj pre verziu úlohy, v ktorej sa môžeme posúvať len *hore* prúdom. Takto vypočítané hodnoty si môžeme označiť $C_{i,j}$.
Optimálne riešenie pôvodnej úlohy samozrejme nemusí byť ani jedného z týchto dvoch typov. Už v zadaní sme videli jeden príklad, ktorého optimálne riešenia nutne zahŕňali ryžovanie aj povyše aj poniže začiatočného úseku u. Zamyslime sa teraz, ako takéto riešenia môžu vyzerať.

Pozorovanie 1: Vždy existuje optimálne riešenie, pri ktorom sa na každom navštívenom úseku všetko ryžovanie udeje ako jeden súvislý časový interval, a to akonáhle prvýkrát navštívime daný úsek.
Dôkaz: Akékoľvek iné riešenie vieme preusporiadať do vyššie uvedenej podoby bez toho, aby sme zmenili celkový čas jeho trvania a celkové množstvo získaného zlata. Napr. ak máme akékoľvek riešenie, pri ktorom na nejakom úseku 3 minúty ryžujeme, potom ho opustíme a neskôr sa tam vrátime a znova 2 minúty ryžujeme, tak môžeme namiesto toho pri prvej návšteve ryžovať 3+2 minút a pri druhej vôbec. Dokopy tak spravíme presne tú istú sadu akcií, takže aj celkový čas aj celkové množstvo získaného zlata ostanú rovnaké.

Pozorovanie 2: Vždy existuje optimálne riešenie, pri ktorom nanajvýš raz zmeníme smer, ktorým sa hýbeme po rieke.
Dôkaz: Každé riešenie má nejaké minúty, kedy sa hýbeme, a nejaké minúty, kedy ryžujeme. Zoberme ľubovoľné riešenie a pozrime sa na množinu úsekov, ktoré sme počas neho navštívili. To, koľko najviac vieme získať ryžovaním, je zjavne jednoznačne určené tým, na ktorých úsekoch vieme ryžovať, a tým, koľko času stratíme pohybmi. Ak vieme tú istú množinu úsekov navštíviť menej pohybmi, ostane nám viac minút na ryžovanie, a teda vieme získať aspoň toľko isto zlata ako keď sme mali času menej – vieme spraviť to isté, čo predtým, a možno potom ešte niekde vyryžovať ďalšie zlato.
Určite teda stačí hľadať optimálne riešenie medzi takými riešeniami, ktoré nejakú množinu úsekov rieky navštívia najmenším možným počtom pohybov.
Keďže sa hýbeme len medzi susednými úsekmi, navštívený úsek rieky vždy tvorí nejaký súvislý interval. Označme u_1 najmenšie a u_2 najväčšie číslo navštíveného úseku. Optimálne riešenie, ktoré u_1 navštívi skôr ako u_2, zjavne vyzerá tak, že ideme zo začiatočného úseku u najskôr proti prúdu ku u_1 a odtiaľ po prúde ku u_2. No a ak navštívime u_2 skôr ako u_1, optimálne riešenie vyzerá presne naopak.
(Dokonca vždy vieme povedať aj to, ktorá z týchto dvoch možností je lepšia – stačí sa pozrieť na to, ktorý z úsekov u_1 a u_2 je ďalej od začiatočného úseku u. Toto pozorovanie však nepotrebujeme využiť, vyskúšať obe možnosti a vybrať lepšiu je rovnako efektívne.)

Optimálne riešenie, pri ktorom nezmeníme smer, už vieme nájsť pomocou hodnôt B a C. Ako nájsť optimálne riešenie s jednou zmenou smeru? Takéto riešenie si vieme rozdeliť na dve fázy. V prvej sa prejdeme jedným smerom a potom sa vrátime späť na začiatočný úsek. V druhej fáze potom z u odídeme opačným smerom ako v prvej a už smer nezmeníme.
Predstavme si, že sme sa už rozhodli, ktorým smerom pôjdeme v prvej fáze a tiež, že v nej strávime presne $x < t$ minút. Akonáhle sme spravili tieto dve rozhodnutia, vieme už vyriešiť každú fázu zvlášť. Presnejšie, môžeme zistiť, koľko najviac zlata vieme dostať za x minút prvej fázy a k tomu pripočítať najväčšie množstvo zlata získateľné za $t - x$ minút druhej fázy.
Na optimálne riešenie druhej fázy vieme využiť už predpočítané hodnoty B a C. Na optimálne riešenie prvej fázy si predpočítame podobné hodnoty B' a C'. Tieto vypočítame pomocou rovnakých vzťahov ako pôvodné B a C, len tentokrát na každý presun zarátame nie jednu ale až dve minúty (keďže každý presun musíme spraviť aj tam aj späť).
Keď už máme tieto hodnoty predpočítané, tak vieme, že optimálne riešenie, ktoré v prvej fáze strávi x minút, počas nich získa $B'_{u,x}$, resp. $C'_{u,x}$ gramov zlata (podľa smeru, ktorým najskôr ideme). Následne spravíme krok na úsek, kde sme ešte neboli, a zvyšok času strávime ryžovaním a pohybmi týmto novým smerom. Počas tejto časti riešenia teda vieme získať nanajvýš $C_{u-1,t-x-1}$, resp. $B_{u+1,t-x-1}$ gramov zlata.
Toto záverečné skúšanie možností vieme spraviť v čase $O(t)$, čo je zanedbateľné oproti predchádzajúcim výpočtom. Celková časová zložitosť nášho riešenia je preto $O(nmt)$. Pamäťová zložitosť je $O(nt)$. Tá by sa dala ďalej zlepšiť na $O(t)$, ale túto drobnú optimalizáciu sme už nevyžadovali.

Listing programu (C++)

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

int t, n, m, u;
vector< vector<int> > Z;

vector<int> dp(int usek, int smer, int krok=1) {
    // zaklad nasej odpovede je ze ryzujeme len tu
    vector<int> odpoved = Z[usek];
    odpoved.resize( t+1, Z[usek].back() );
    // ak uz sme na kraji rieky, to je vsetko
    if (!(0 <= usek+smer && usek+smer < n)) return odpoved;
    // zistime odpovede pre susedny usek
    vector<int> ostatne = dp(usek+smer, smer, krok);
    // prepocitame ako najlepsie zapojit zvysok rieky
    for (int c=1; c<=t; ++c) for (int tu=0; tu<=m && tu<=c; ++tu) {
        // najdeme opt. riesenie ak z "c" minut "tu" ryzujeme tu
        int toto = Z[usek][tu], ostava = c-tu-krok;
        if (ostava > 0) toto += ostatne[ostava];
        odpoved[c] = max( odpoved[c], toto );
    }
    return odpoved;
}

int main() {
    // nacitame vstup, spravime prefixovy sucet Z pre kazdy usek rieky
    cin >> t >> n >> m >> u;
    --u; // v programe cislujeme useky od 0
    Z.resize(n);
    for (int i=0; i<n; ++i) {
        Z[i].push_back(0);
        for (int j=0; j<m; ++j) { int z; cin >> z; z += Z[i].back(); Z[i].push_back(z); }
    }
    // vypocitame polia popisane v texte riesenia
    vector<int> B=dp(u,1), C=dp(u,-1), B2=dp(u,1,2), C2=dp(u,-1,2);
    vector<int> B3(t+1,0), C3(t+1,0); // verzie poli B a C zacinajuc od susednych usekov
    if (u < n-1) B3=dp(u+1,1);
    if (u > 0) C3=dp(u-1,-1);
    // najdeme globalne optimum vyskusanim rieseni s najviac jednou zmenou smeru
    int optimum = max(B[t], C[t]); // ak nezmenime smer
    for (int c=0; c<t; ++c) optimum = max( optimum, max( B2[c]+C3[t-c-1], C2[c]+B3[t-c-1] ) );
    cout << optimum << endl;
}
```

ŠTYRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Forišek
Recenzia: Michal Anderle, Tomáš Belan, Paulína Smolárová
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2025