A-III-4 Bandasky

Najmenší počet bandasiek, pre ktorý riešenie vždy existuje, nájdeme tak, že nájdeme situáciu s najviac bandaskami, pre ktorú ešte riešenie neexistuje, a ich počet zväčšíme o 1. Chceme teda do stromu rozmiestniť čo najviac bandasiek tak, aby Jafar nevedel žiadnu dostať do oázy j.
Predstavíme si, že vrchol j je koreňom nášho stromu a všetky diaľnice sú orientované smerom k vrcholu j. Ľahko nahliadnemem, že Jafarovi sa nikdy neoplatí žiadnu bandasku posielať po diaľnici opačným smerom, teda preč od j. (Ak by sme z toho smeru následne nikdy nedostali bandasku naspäť, zbytočne sme si minuli bandasku. A ak sme niekedy v budúcnosti nejakú dostali, mohli sme vynechať dve akcie – tú, kde posielame túto bandasku tam, a tú, kde posielame neskôr bandasku späť – a mali by sme rovnako dobré riešenie plus ušetrené dve bandasky.)
A aj smer ku j je následne priamočiary: kedykoľvek, keď máme v hociktorej oáze (aspoň) dve bandasky, nič nepokazíme, keď jednu z nich pošleme o krok bližšie ku j – nič iné s nimi už aj tak nevieme robiť.
Tu si môžeme všimnúť, že každý krok smerom preč od cieľa nám zdvojnásobuje potrebný počet bandasiek. Majme napr. cestu $1 - 2 - 3 - 4$. Ak máme v oáze 1 dve bandasky, vieme jednu z nich dostať do oázy 2. Ak máme v oáze 1 štyri bandasky, vieme spraviť predchádzajúcu akciu dvakrát, čím dostaneme v oáze 2 dve bandasky a teda získame možnosť jednu z nich poslať do oázy 3. A podobne o krok ďalej: z osem bandasiek v oáze 1 budú štyri v oáze 2, potom dve v oáze 3 a z nich jedna v oáze 4.
Toto vieme popísať aj formálnejšie. Pozrime sa na situáciu na konci a všimnime si ľubovoľnú bandasku x, ktorá ešte stále existuje v nejakom vrchole v. Teraz sa pozrime na situáciu na začiatku a všimnime si všetky bandasky, ktoré sme zapojili do toho, aby sa bandaska x dostala do svojho cieľa. Pre každú z týchto štartových bandasiek (vrátane x samotnej) sa pozrime na jej vzdialenosť od vrcholu v. Každej štartovej bandaske, ktorá je vo vzdialenosti d od cieľového vrcholu v, priradíme skóre $1/2^d$. Teraz tvrdíme, že súčet skóre všetkých bandasiek je vždy presne rovný jednej.
Dôkaz spravíme ľahko matematickou indukciou od počtu bandasiek: Ak je bandaska sama, je už v cieli a má skóre $1/2^0 = 1/1 = 1$. No a ak kedykoľvek máme dve bandasky v ľubovoľnej vzdialenosti $d + 1$ a vyrobíme z nich jednu bandasku vo vzdialenosti d, celkový súčet skóre sa tým nezmení.

Všeobecnejšia otázka

Zamyslime sa teraz nad trochu všeobecnejšou otázkou: Majme zakorenený strom s koreňom x, ktorý má deti $y_1, \ldots, y_k$. Koľko najviac bandasiek môžeme v tomto strome rozmiestniť tak, aby ich na konci v koreni skončilo nanajvýš z?

Ak x už nemá žiadne deti ($k = 0$), odpoveď je zjavne z.

Nech teraz $k > 0$, teda x má nejaké deti. Pre každý z podstromov s koreňmi $y_1, \ldots, y_k$ sa pozrime na jeho hĺbku – teda na maximálnu vzdialenosť medzi x a listom v tomto podstrome. Bez ujmy na všeobecnosti nech y_1 je vrchol s najhlbším podstromom a nech ℓ je jeden konkrétny najhlbší list v tomto podstrome, vo vzdialenosti d od vrcholu x. Potom tvrdíme nasledovné: existuje optimálne rozmiestnenie bandasiek, pri ktorom v x skončí práve z bandasiek a všetky tam prídu z vrcholu y_1.

Dôkaz je zjavný: Zoberme ľubovoľnú bandasku, ktorá skončila v koreni. Vieme, že bandasky, z ktorých vznikla, mali dokopy skóre 1, a keďže najväčšia možná vzdialenosť od koreňa je d, najmenšie možné skóre jednej bandasky na začiatku je $1/2^d$, a teda jednou z optimálnych možností je zjavne tá, kde bandaska v koreni vznikla z 2^d bandasiek, ktoré všetky začínali v liste ℓ.

Algoritmus

Z práve dokázaného tvrdenia teda vyplýva, že optimálne rozmiestnenie bandasiek, ktoré hľadáme v predchádzajúcej časti, vieme zostrojiť nasledovne:
Ak x (koreň stromu) má deti, nech y_1 je jeho dieťa s najhlbším podstromom. Potom:

- Samotný koreň bude na začiatku prázdny.

- V podstrome s koreňom y_1 chceme bandasky rozmiestniť tak, aby ich do y_1 prišlo nanajvýš $2z+1$. (To je najväčší počet bandasiek, pre ktorý ich potom odtiaľ do koreňa vieme poslať len z.)
- V každom z ostatných podstromov chceme bandasky rozmiestniť tak, aby do jeho koreňa y_i prišla nanajvýš jedna.

Na každý z podstromov sa preto rekurzívne zavoláme s príslušnou otázkou.
Ak si na začiatku raz predpočítame hĺbky všetkých podstromov, vieme potom pri tomto algoritme každý vrchol spracovať v čase priamo úmernom jeho stupňu, a keďže súčet stupňov vrcholov v strome je $2n-2$ (každá hrana je započítaná v dvoch stupňoch vrcholov), má náš algoritmus celkovo lineárnu časovú zložitosť.

Listing programu (C++)

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

const long long MOD = 1000000007;

int N, J;
vector< vector<int> > strom;
vector<int> rodic, hlbka, najhlbsi_syn;
long long odpoved;

void dfs(int kde, int odkial = -1) {
    rodic[kde] = odkial;
    for (int kam : strom[kde]) if (kam != odkial) {
        dfs(kam, kde);
        if (hlbka[kam] + 1 > hlbka[kde]) {
            hlbka[kde] = hlbka[kam] + 1;
            najhlbsi_syn[kde] = kam;
        }
    }
}

void vypln(int kde, long long kolko) {
    if (hlbka[kde] == 0) {
        // sme v liste
        odpoved = (odpoved + kolko) % MOD;
    } else {
        // tlacime dalej
        vypln( najhlbsi_syn[kde], (2*kolko+1)%MOD );
        for (int kam : strom[kde]) {
            if (kam == rodic[kde]) continue;
            if (kam == najhlbsi_syn[kde]) continue;
            vypln( kam, 1 );
        }
    }
}

int main() {
    cin >> N >> J;
    strom.resize(N);
    for (int n=0; n<N-1; ++n) {
        int x, y;
        cin >> x >> y;
        strom[x].push_back(y);
        strom[y].push_back(x);
    }
    rodic.resize(N, -1);
    hlbka.resize(N, 0);
    najhlbsi_syn.resize(N, -1);
    dfs(J);
    odpoved = 0;
    vypln(J, 0);
    cout << ((1+odpoved)%MOD) << endl;
}
```

A-III-5 Práčka

Pre $k=0$ má pre každé $i<j$ platiť $a_j - a_i \leq 0(j-i) = 0$, čiže $a_i \geq a_j$. Inými slovami, chceme vyrobiť nerastúcu postupnosť.
Keď si zvolíme, ktoré pozície zmeniť, tie nezmenené musia tvoriť nerastúcu podpostupnosť pôvodnej postupnosti. A naopak, keď si zvolíme ľubovoľnú nerastúcu podpostupnosť, vždy vieme zmeniť všetky ostatné prvky tak, aby celý výsledok bol nerastúci. Riešením je teda nájsť jednu najdlhšiu nerastúcu podpostupnosť a zmeniť zvyšok.

Oprava postupnosti na nerastúcu

Predstavme si, že sme našli jednu konkrétnu najdlhšiu nerastúcu podpostupnosť. Všetky prvky, ktoré do nej nepatria, prelepíme nálepkami. Na tieto nálepky chceme teraz napísať nové hodnoty tak, aby celá nová postupnosť bola nerastúca.
Ak postupnosť začína nálepkou, napíšeme na ňu najväčšiu hodnotu (teda napr. prvú neprelepenú, alebo pokojne rovno 10^9), tým určite nič nepokazíme.
No a potom už stačí prejsť zvyšok postupnosti zľava doprava a na každú ďalšiu nálepku, ktorú stretneme, napísať rovnaké číslo ako je tesne pred ňou. Výsledkom tohto procesu je zjavne nerastúca postupnosť a počet zmien, ktoré sme spravili, je zjavne optimálny.

Nájdenie najdlhšej nerastúcej podpostupnosti

Toto vieme spraviť v čase $O(n \log n)$ napr. nasledovne:
Predstavme si, že sme už spracovali nejaký kus vstupnej postupnosti, napríklad $(10, 17, 12, 14, 11, 16, 14, 1, 12, 15)$. Z tejto postupnosti vieme vybrať veľa rôznych dvojprvkových nerastúcich postupností, napr. $(10, 5)$, $(16, 16)$, alebo $(16, 14)$.
Teraz nám na vstupe príde nový prvok a_x. Predstavme si, že chceme vyrobiť trojprvkovú nerastúcu podpostupnosť, ktorá končí týmto a_x. Toto musíme spraviť tak, že zoberieme niektorú dvojprvkovú nerastúcu podpostupnosť, ktorú sme mali v už spracovaných dátach, a na jej koniec pridáme a_x. No a ktorá zo všetkých možných dvojprvkových podpostupností je na to najvhodnejšia? Je zjavné, že ak je napríklad postupnosť $(10, 5, a_x)$ nerastúca, tak aj postupnosť $(16, 14, a_x)$ bude nerastúca, lebo ak $a_x \leq 5$, tak tým skôr platí aj $a_x \leq 14$. Inými slovami, najlepšia je tá postupnosť, ktorá končí najväčším možným číslom. Každé a_x, ktoré by pasovalo za nejakú inú podpostupnosť, bude pasovať aj za túto.
Budeme si teda udržiavať hodnoty c_j s nasledovným významom: keď sa pozrieme na všetky možné nerastúce j-prvkové podpostupnosti v už spracovaných dátach a zoberieme posledný prvok každej z nich, najväčšia z takto získaných hodnôt bude práve c_j. Špeciálne budeme mať $c_0 = \infty$ (za postupnosť dĺžky 0 sa dá pridať čokoľvek) a $c_j = -\infty$ ak ešte v spracovaných dátach žiadna nerastúca j-prvková podpostupnosť neexistuje.
Príklad: ak sme už spracovali $(10, 17, 12, 14, 11, 16, 14, 1, 12, 15)$, tak budeme mať $c_1 = 17$, $c_2 = 16$, $c_3 = 15$, $c_4 = 12$ a od c_5 ďalej budú všetky mať hodnotu $-\infty$.
Všimnite si, že rôzne c_i budú vo všeobecnosti zodpovedať rôznym postupnostiam. Napr. tu c_3 zodpovedá postupnosti $(17, 16, 15)$, zatiaľ čo c_4 je koniec postupnosti $(17, 16, 14, 12)$.
Ku každej c_i si môžeme navyše pamätať aj index d_i, na ktorom táto hodnota ležala. V našom príklade by sme (používajúc indexovanie od nuly) mali $d_1 = 1$, $d_2 = 5$, $d_3 = 9$ a $d_4 = 8$.
Iný príklad: ak sme už spracovali $(7, 7, 7)$, máme $c_1 = c_2 = c_3 = 7$, $c_4 = -\infty$ a indexy $d_1 = 0$, $d_2 = 1$ a $d_3 = 2$.

Nasledujúce pozorovanie, ktoré použijeme: hodnoty c_i sú vždy usporiadané podľa veľkosti v **nerastúcom** poradí. Napríklad vždy platí $c_3 \geq c_4$, lebo keď zoberiem *najlepšiu* nerastúcu podpostupnosť dĺžky 4 a odstránim z nej posledný prvok, tak dostanem *nejakú* (dokonca nie nutne najlepšiu) nerastúcu podpostupnosť dĺžky 3. A keďže pôvodná postupnosť končila hodnotou c_4, táto z nej vyrobená končí hodnotou väčšou alebo rovnou c_4.
Predstavme si teraz, že prečítame zo vstupu nasledujúci prvok a_x a zaujíma nás, aká najdlhšia nerastúca podpostupnosť končí týmto prvkom. Aby sme to zistili, chceme nájsť najväčšie i také, že $c_i \geq a_x$. Potom je zjavné, že najdlhšia nerastúca podpostupnosť končiaca práve prečítaným a_x má dĺžku $i + 1$.
No a keďže vieme, že hodnoty c sú usporiadané podľa veľkosti, vieme hľadané i nájsť v logaritmickom čase binárnym vyhľadávaním. Navyše si po nájdení správneho i vieme pre pozíciu x zapamätať, že najlepšia postupnosť končiaca tu vznikla predĺžením najlepšej postupnosti končiacej na pozícii d_i. Z takto zapamätaných indexov vieme potom pre každú pozíciu jednu najdlhšiu na nej končiacu podpostupnosť efektívne zostrojiť.

Ostáva nám posledný krok: zistiť, ako sa hodnoty c_i zmenia, keď na koniec spracovanej postupnosti pridáme práve prečítanú hodnotu a_x.
Ukáže sa, že zmena je vždy len minimálna. Spomeňme si, že v predchádzajúcom kroku sme našli najväčšie i také, že $c_i \geq a_x$. Teraz platí:

- Pre každé $j \leq i$ už aj bez a_x existuje j-prvková nerastúca podpostupnosť končiaca prvkom väčším alebo rovným a_x. Hodnoty c_1 až c_i sa teda nezmenia.
- Nemáme žiaden spôsob, ako vyrobiť nerastúcu podpostupnosť dĺžky $i+2$ a viac, ktorá by končila práve prečítaným x. Ani hodnoty od c_{i+2} ďalej sa teda nezmenia.
- Tým pádom jediné, čo sa zmení, je hodnota c_{i+1}. Doterajšie c_{i+1} bolo ostro menšie ako a_x, odteraz zjavne máme $c_{i+1} = a_x$.

Napríklad ak by sme pokračovali vo vyššie uvedenom príklade tým, že prečítame zo vstupu ako ďalší prvok postupnosti hodnotu $a_x = 13$, zmenila by sa hodnota c_4 z 12 na 13. Ak by sme namiesto toho prečítali $a_x = 12$, ostala by $c_4 = 12$ a zmenila by sa c_5 z $-\infty$ na 12.

Po prečítaní a spracovaní n prvkov budeme mať nanajvýš n-prvkovú najdlhšiu nerastúcu podpostupnosť, preto je v ľubovoľnom okamihu len $O(n)$ prvkov postupnosti c ktoré majú konečnú veľkosť. A teda binárne vyhľadávanie v nich beží v čase $O(\log n)$. Každý prvok zo vstupu teda prečítame a spracujeme v logaritmickom čase, a tým pádom je celková časová zložitosť tohto riešenia $O(n \log n)$.

Všeobecné riešenie

Podmienku, že pre všetky $i < j$ má platiť $(a_j - a_i) \leq k(j - i)$, môžeme upraviť do ekvivalentnej podoby: $(a_i - k \cdot i) \geq (a_j - k \cdot j)$.
Uvažujme teraz postupnosť $b_i := a_i - k \cdot i$. Táto postupnosť má pre všetky $i < j$ spĺňať $b_i \geq b_j$, čiže má byť nerastúca. To je ale presne úloha, ktorú už vieme riešiť. Stačí teda použiť na postupnosť b vyššie popísané riešenie, a potom z opravenej postupnosti b dopočítať opravenú postupnosť a.

Ostáva posledný technický detail: Zadanie vyžaduje, aby všetky hodnoty opravenej a_i ležali v rozsahu od 1 po 10^9. Na to stačí, aby sme po vyššie popísanej úprave (spravenej v neohraničených celých číslach, resp. 64-bitových premenných) všetky nové hodnoty, ktoré vyšli väčšie ako 10^9, zmenili na rovné 10^9.
Hodnoty po oprave vyššie popísaným spôsobom sú zjavne naďalej všetky kladné. Hodnoty, ktoré nemeníme, všetky ostávajú v povolenom rozsahu, takže nad 10^9 môžu vybehnúť len doplňané nové hodnoty. No a rozborom prípadov ľahko overíme, že tým, že novým hodnotám nedovolíme prekročiť 10^9, nemôžu vzniknúť žiadne zlé dvojice indexov $i < j$.

Listing programu (C++)

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

const long long INF = 1LL << 40;
const long long MAX_ALLOWED = 1000000000;

int main() {
    int N, T;
    long long K;
    cin >> N >> K >> T;
    vector<long long> A(N);
    for (auto &a : A) cin >> a;

    for (int i=0; i<N; ++i) A[i] -= K*i;

    vector<long long> best(1, -INF);
    vector<int> best_index(1, -1), prev(N, -1);

    for (int i=0; i<N; ++i) {
        int lo = 0, hi = best.size();
        while (hi - lo > 1) {
            int med = (lo + hi) / 2;
            if (best[med] >= A[i]) lo = med; else hi = med;
        }
        best.push_back(INF); best_index.push_back(-1);
        prev[i] = best_index[hi-1];
        best[hi] = A[i];
        best_index[hi] = i;
        if (best.back() == INF) { best.pop_back(); best_index.pop_back(); }
    }
    int longest = best.size() - 1;
    cout << (N - longest) << endl;
    if (T == 1) return 0;
```

```
    for (int i=0; i<N; ++i) A[i] += K*i;

    vector<bool> keep(N, false);
    int where = best_index.back();
    while (true) {
        if (where == -1) break;
        keep[where] = true;
        where = prev[where];
    }

    int start = 0;
    while (!keep[start]) ++start;
    for (int i=0; i<start; ++i) A[i] = A[start];
    for (int i=start; i<N; ++i) if (!keep[i]) A[i] = min( A[i-1]+K, MAX_ALLOWED );
    for (int i=0; i<N; ++i) cout << A[i] << (i+1==N ? "\n" : "␣");
}
```

A-III-6 Šachisti

Ratingy tréningom len rastú, preto nikomu nikdy nesmieme zvýšiť rating nad jeho želaný. Špeciálne teda platí, že ak niekoho začiatočný rating r_i je ostro väčší ako jeho želaný rating c_i, riešenie neexistuje. Toto môžeme na začiatku skontrolovať a potom vo zvyšku riešenia predpokladať, že pre každého platí $r_i \leq c_i$.

Uvažujme graf, ktorého vrcholy sú šachisti a hrany sú priateľstvá medzi nimi.
Predstavme si, že chceme dosiahnuť, aby konkrétny človek x_1 dostal rating y. Na to zjavne v prvom rade musí existovať iný človek x_2, ktorý momentálne presne tento rating má. To ale samozrejme nestačí, ešte musíme aj vedieť tento rating postupnými tréningami dostať k človeku x_1 – teda musí existovať v našom grafe nejaká cesta, po ktorej sa rating y vie dostať k človeku x_1.
Ktorí šachisti môžu ležať na tejto ceste? Len tí, ktorí môžu niekedy mať rating y. Tu si musíme dať pozor na dve veci. V prvom rade platí, že ak šachista začína s ratingom $r_i > y$, nikdy nevie mať rating y. Taktiež ale platí, že ak má šachista skončiť s ratingom $c_i < y$, nesmieme ho nikdy natrénovať na vyšší rating y.
V našom grafe *platnou cestou* pre človeka x_1 bude preto cesta, ktorá spĺňa nasledovné podmienky:

1. Končí človekom x_1, ktorého želaný rating je $c_{x_1} = y$.
2. Začína človekom x_2, ktorého začiatočný rating je $r_{x_2} = y$.
3. Na celej ceste nie je žiaden šachista, ktorý má začiatočný rating väčší ako y.
4. Na celej ceste nie je žiaden šachista, ktorý má želaný rating menší ako y.

Malo by byť zjavné, že ak pre nejakého človeka neexistuje platná cesta, tak nevieme nijak dosiahnuť, aby dostal svoj želaný rating, a teda riešenie neexistuje.
Tiež by malo byť zjavné, že keď takúto cestu nájdeme, tak vieme dosiahnuť, aby tento konkrétny jeden človek x_1 dostal svoj želaný rating: postupne pozdĺž cesty sa všetci naň natrénujú.
Použitím takejto cesty samozrejme zmeníme viacerým šachistom ratingy a to môže ovplyvniť existenciu podobných ciest pre iných šachistov. My ale tvrdíme, že ak to budeme robiť šikovne, všetko bude fungovať.

Tvrdenie: Celkové riešenie existuje práve vtedy, ak v pôvodnom grafe pre každého šachistu existuje platná cesta.

Dôkaz: Cesty použijeme usporiadané podľa ratingu y, od najmenšieho po najväčší.
Pre každého človeka sa pozrime najskôr na okamih, kedy ideme použiť jeho cestu na to, aby on dostal svoj želaný rating y. Doteraz sme používali len cesty s ratingom $\leq y$, každý človek má teda buď svoj začiatočný rating, alebo je jeho rating najviac y. Nevznikli nám teda žiadni noví ľudia s ratingom väčším ako y, a teda našu cestu naozaj aj teraz môžeme použiť: každý šachista, ktorý na nej mohol byť na začiatku, na nej stále môže byť.
No a teraz si uvedomme, že akonáhle začneme používať cesty s ratingom väčším ako je rating y tohto človeka, tieto cesty nemôžu viesť cez neho (už na začiatku pre ne nespĺňal poslednú podmienku), a teda jeho rating už ostane y až do konca.

Ľahšie špeciálne typy grafov

Na kompletnom grafe je riešenie našej úlohy ľahké: Stačí, aby každý z cieľových ratingov c_i existoval v množine začiatočných ratingov r_i. Totiž keď sa každá dvojica priateľí, tak ako každú cestu vieme zobrať priamu hranu.

Na hviezde všetky cesty vedú cez stred a len ten musíme kontrolovať.

Na dlhej ceste si ľahko uvedomíme, že keď chceme dostať rating y šachistovi x, stačí uvažovať dve možnosti: buď ho tam dostaneme od *najbližšieho* šachistu so začiatočným ratingom y naľavo od x, alebo od najbližšieho takého napravo. Stačí teda pre tieto dve cesty skontrolovať, či sú platné. No a toto overenie vieme previesť na otázky na minimum a maximum súvislého úseku v postupnosti, a na také otázky vieme efektívne odpovedať napr. tak, že si nad tou postupnosťou postavíme intervalový strom.

Na malých grafoch (či už stromoch alebo všeobecných) si vieme dovoliť z každého človeka spustiť jedno prehľadávanie grafu, a tým overiť, či pre neho existuje nejaká cesta. Respektíve by stačilo prehľadať celý graf raz pre každú hodnotu cieľového ratingu, ale asymptotickú časovú zložitosť najhoršieho prípadu nám táto optimalizácia nezmení.

Veľké stromy

Implementovaním riešení pre vyššie popísané triedy grafov sa dalo dokopy získať 8 bodov (z čoho 6 za riešenie pre malé grafy). Ostávajú posledné dva body. V tejto časti si ukážeme riešenie, ktoré získa ten deviaty tým, že vyrieši sadu 8: veľké stromy, v ktorých sú začiatočné ratingy r_i navzájom rôzne.

Pre každého šachistu x_1 existuje v celom strome nanajvýš jeden šachista x_2, ktorý má na začiatku jeho želaný rating. Tým je určená celá cesta pre x_1, ostáva len skontrolovať, či je platná.
Cestu z x_1 do x_2 si vždy vieme rozdeliť na dve (možno prázdne) cesty x_1x_3 a x_2x_3 tak, aby obe išli v strome len dohora. Vrchol x_3, kde cestu rozdelíme, je najbližším spoločným predkom x_1 a x_2.
Stačí teda, ak pre každú cestu dohora stromom vieme povedať, aký najväčší začiatočný a aký najmenší želaný rating na nej leží. Na toto si vieme predpočítať v čase $O(n \log n)$ užitočné údaje nasledovne: pre každý vrchol x stromu a každé i (do $i = \log_2 n$) si spočítame toto maximum a minimum pre cestu, ktorá ide z x dohora a má dĺžku 2^i.
Podobné predpočítané údaje vieme využiť aj na efektívne nájdenie najbližšieho spoločného predka. Viac detailov o tomto algoritme nájdete tu: https://www.ksp.sk/kucharka/lca/.
Každú cestu takto vieme celú skontrolovať v čase $O(\log n)$. Toto riešenie má teda časovú aj pamäťovú zložitosť $O(n \log n)$.

Iné efektívne riešenie vieme spraviť pomocou tzv. heavy-light dekompozície stromu (https://en.wikipedia.org/wiki/Heavy-light_decomposition). Základná myšlienka je taká, že ak je strom košatý a plytký, tak sme spokojní, lebo všetky cesty sú krátke a teda môžeme na otázky odpovedať hrubou silou. Vadia nám len stromy, ktoré sa málo vetvia a sú hlboké. Pri vyššie spomenutej dekompozícii sa ukáže, že vieme v ľubovoľnom strome nájsť malý počet dlhých ciest tak, že celý zvyšok ostane košatý. Pre každú cestu zvlášť potom použijeme vyššie popísané riešenie pre cestu, no a zvyšok stromu si už môžeme dovoliť spracovať hrubou silou.
Takéto riešenie každého šachistu skontroluje v čase $O(\log^2 n)$, a teda jeho celková časová zložitosť je $O(n \log^2 n)$. Pamäťová zložitosť je len $O(n)$.

Veľké všeobecné grafy

Celý text odtiaľto až po koniec týchto vzorových riešení popisuje, ako získať posledný bod za túto úlohu :)
Zoberme si konkrétny želaný rating y. Teraz si predstavme, že sme zmazali vrcholy, ktoré nemôžu byť na ceste pre tento rating. Ostanú nám nejaké komponenty súvislosti. Kedy existujú cesty pre šachistov, ktorí chcú tento rating? Zjavne práve vtedy, keď pre každý komponent platí: „ak obsahuje niekoho so želaným ratingom y, musí obsahovať aj niekoho so začiatočným ratingom y“.

Pozrime sa teraz na nasledujúci rating $y + 1$. Ako sa zmení náš graf? Začnú existovať vrcholy, ktoré majú začiatočný rating presne $y + 1$ a prestanú existovať vrcholy, ktoré majú želaný rating presne y.
Malo by byť zjavné, že keď budeme postupne iterovať cez všetky ratingy, tak každý vrchol len raz začne a raz prestane existovať.
Na ratingy, cez ktoré postupne iterujeme, sa môžeme dívať ako na čas. Pre každú hranu nášho grafu si vieme spočítať interval časov, počas ktorých existuje: sú to časy, kedy naraz existujú oba jej koncové vrcholy.

Samotný priebeh kontroly cesty

Na začiatku si predspracujeme vrcholy, aby sme pre ľubovoľné y vedeli efektívne nájsť aj vrcholy s $r_i = y$, aj vrcholy s $c_i = y$.
Predstavme si, že už máme zostrojené vyššie popísané komponenty súvislosti pre nejaké konkrétne y – teda pre každý vrchol vieme efektívne povedať ID komponentu, do ktorého patrí.
Ako overíme, či pre toto y existujú všetky potrebné cesty? Stačí najskôr do prázdnej množiny pre každý vrchol s $c_i = y$ vložiť ID jeho komponentu, potom pre každý vrchol s $r_i = y$ z tej množiny odstrániť ID jeho komponentu (ak tam je), a na záver skontrolovať, či množina ostala prázdna.
Za celý beh riešenia bude každý vrchol raz takto vložený a (nanajvýš) raz odstránený, takže dokopy bude čas strávený týmito kontrolami zanedbateľný oproti zvyšku riešenia.

Lenivejšie „hackerské“ riešenie

Pre každý prechod z času=ratingu y na $y+1$ sa môžeme pozrieť na to, koľkým hranám sa zmení stav. Začínajúc od času 1 si teraz môžeme časovú os nasekať na kusy nasledovne: postupne zvyšujeme koniec aktuálneho úseku a počítame si zmeny stavu hrán, ktoré sme videli, až kým neprídeme na prechod, vrátane ktorého by už tento počet prekročil $\sqrt{m}$. Tam ukončíme aktuálny kus a od nasledujúceho ratingu začneme ďalší. Takto dostaneme $O(\sqrt{m})$ kusov a vnútri každého sa udeje nanajvýš $\sqrt{m}$ zmien stavu hrany. (Na niektorých konkrétnych prechodoch, ktoré skončili na hraniciach medzi úsekmi, sa môže diať takýchto zmien aj rádovo viac. Tieto zmeny ale nikdy nebudeme spracúvať!)
Pre každý interval časov teraz spravíme nasledovné: Zostrojíme si množinu hrán, ktoré existujú počas celého toho intervalu, a druhú malú množinu hrán, ktoré počas neho menia stav. Raz spravíme prehľadávanie (alebo Union-Find) na prvej množine hrán a nájdeme si komponenty súvislosti, ktoré im zodpovedajú. Potom zvlášť spracujeme každý relevantný čas (t.j. taký, ktorý zodpovedá niekoho želanému ratingu) v danom intervale.
Spracovanie konkrétneho času y vyzerá nasledovne:

1. Prejdeme $O(\sqrt{m})$ hrán, ktoré menia stav počas daného intervalu, a zistíme, ktoré existujú v čase y.
2. Pridáme tieto hrany do grafu a v čase približne priamo úmernom ich počtu určíme nové komponenty.
3. Keď poznáme tie, skontrolujeme všetky vrcholy, ktoré chcú skončiť na tomto y.

Takéto riešenie bude mať asymptotickú časovú zložitosť niekde v okolí $m\sqrt{m}$. Presná časová zložitosť závisí od konkrétnej zvolenej implementácie zostrojovania komponentov súvislosti a použitých dátových štruktúr.

Efektívnejšie riešenia

Existujú aj riešenia, ktoré pre všeobecné grafy bežia dokonca v čase $O(m \log n)$. Ak sa chcete nad nejakým zamyslieť, pre inšpiráciu uvedieme napr. odkaz na link/cut stromy: https://en.wikipedia.org/wiki/Link/cut_tree.
My si ukážeme myšlienkovo aj implementačne jednoduchšie riešenie, ktoré bude mať jeden logaritmus navyše – pobeží teda v čase $O(m \log^2 n)$.
Predstavme si intervalový strom nad časovou osou. Pre každú hranu grafu si môžeme nájsť interval časov, počas ktorých existuje, a potom tento interval vložiť do tohto intervalového stromu. Tým dostaneme pre každú hranu $O(\log n)$ vrcholov intervalového stromu, ktoré dokopy zodpovedajú jej celému intervalu existencie. V každom z týchto vrcholov si túto hranu poznačíme.
Teraz prehľadáme do hĺbky celý tento intervalový strom. Vždy, keď vojdeme do vrcholu, tak v ňom zapamätané hrany začnú existovať, a vždy, keď vrchol opustíme, tak existovať prestanú. Zjavne teda vždy, keď prídeme do

listu (t.j. vrcholu predstavujúceho jeden konkrétny čas y), budeme mať práve zostrojenú správnu množinu hrán: práve všetky hrany, ktoré existujú v tomto čase y. Môžeme teda teraz skontrolovať šachistov, ktorí chcú skončiť s ratingom y.

No a ako si počas tohto celého udržiavať komponenty, aby sme tieto kontroly vedeli robiť efektívne? Budeme opäť raz robiť Union-Find, tentokrát špecificky s heuristikou „union by rank“ ale bez kompresie ciest, aby sme pri každej operácii Union spravili len jednu zmenu v pamäti. (Takýto Union-Find má zaručenú časovú zložitosť $O(\log n)$ na operáciu.) Vždy, keď pridávame hranu a robíme nejakú zmenu v pamäti pre Union-Find, uložíme si prepísané hodnoty na zásobník. No a vždy, keď chceme nejakú hranu odstrániť, z vrchu zásobníka zoberieme prepísané hodnoty a vrátime ich späť.

Pre každý z $O(n \log n)$ „kusov hrán“ spravíme konštantný počet operácií s dátovou štruktúrou Union-Find, čím dostávame sľubovanú časovú zložitosť.

Listing programu (C++)

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

const string ANO = "ANO", NIE = "NIE";

int N, M, MAXR;
vector<pair<int,int> > E;
vector<int> R, C;
map<int, vector<int> > ponuka, potrebuje;

struct zmena { int x, y, byvaly_rank_x; };

struct union_find {
    vector<int> otec, rank;
    vector<zmena> undo_log;
    union_find(int N) { otec.resize(N); iota(otec.begin(),otec.end(),0); rank.resize(N); }
    int sef(int x) { if (x==otec[x]) return x; else return sef(otec[x]); }
    bool spoj(int x, int y) {
        x = sef(x); y = sef(y); if (x == y) return false;
        if (rank[x] < rank[y]) swap(x,y);
        undo_log.push_back( { x, y, rank[x] } );
        otec[y] = x;
        rank[x] = max( rank[x], rank[y]+1 );
        return true;
    }
    void undo(int pokial) {
        while (int(undo_log.size()) > pokial) {
            zmena z = undo_log.back(); undo_log.pop_back();
            rank[z.x] = z.byvaly_rank_x;
            otec[z.y] = z.y;
        }
    }
};

struct intervalac {
    int L;
    vector< vector< vector<int> > > data;
    intervalac(int N) {
        data.clear();
        for (L=0; ; ++L) {
            data.push_back( vector< vector<int> >(1<<L) );
            if (int(data.back().size()) >= N+3) break;
        }
    }
    void vloz_interval(int id, int lo, int hi, int wx=0, int wy=0, int wlo=0, int whi=-1) {
        if (whi==-1) whi = 1<<L;
        if (lo <= wlo && whi <= hi) { data[wx][wy].push_back(id); return; }
        if (hi <= wlo || whi <= lo) return;
        vloz_interval(id, lo, hi, wx+1, 2*wy, wlo, (wlo+whi)/2);
        vloz_interval(id, lo, hi, wx+1, 2*wy+1, (wlo+whi)/2, whi);
    }
    bool prejdi(union_find &UF, int, int, int, int);
};

pair<int, int> interval_hrany(int m) {
    // mozno-prazdny uzavrety interval casov, v ktorych existuje hrana m
    return { max( R[E[m].first], R[E[m].second] ), min( C[E[m].first], C[E[m].second] ) };
}

void compress(vector<int> &R, vector<int> &C) {
    set<int> X;
    for (int x : R) X.insert(x);
    for (int x : C) X.insert(x);
    map<int, int> rename;
    int id = 0;
    for (int x : X) rename[x] = ++id;
```

```
    MAXR = id;
    for (int &x : R) x = rename[x];
    for (int &x : C) x = rename[x];
}

bool intervalac::prejdi(union_find &UF, int wx=0, int wy=0, int wlo=0, int whi=-1) {
    if (whi==-1) whi = 1<<L;
    // pridaj hrany, ktore su v aktualnom vrchole
    int aktualny_stav = UF.undo_log.size();
    for (int m : data[wx][wy]) UF.spoj( E[m].first, E[m].second );
    // ak si v liste, spracuj ho, ak nie, rekurzivne sa zavolaj
    bool return_value;
    if (whi - wlo == 1) {
        int rating = wlo;
        set<int> komponenty;
        for (int x : potrebuje[rating]) komponenty.insert( UF.sef(x) );
        for (int x : ponuka[rating]) komponenty.erase( UF.sef(x) );
        return_value = komponenty.empty();
    } else {
        return_value = prejdi(UF, wx+1, 2*wy, wlo, (wlo+whi)/2);
        if (return_value) return_value = prejdi(UF, wx+1, 2*wy+1, (wlo+whi)/2, whi);
    }
    // pri odchode odober hrany
    UF.undo(aktualny_stav);
    return return_value;
}

string vyries() {
    cin >> N >> M;
    R.clear(); R.resize(N); C.clear(); C.resize(N);
    for (int &x : R) cin >> x;
    for (int &x : C) cin >> x;
    compress(R, C);
    E.clear();
    for (int m=0; m<M; ++m) { int x, y; cin >> x >> y; E.push_back({x,y}); }

    for (int n=0; n<N; ++n) if (C[n] < R[n]) return NIE; // nikomu nevie klesnut rating
    ponuka.clear(); potrebuje.clear();
    for (int n=0; n<N; ++n) ponuka[R[n]].push_back(n);
    for (int n=0; n<N; ++n) potrebuje[C[n]].push_back(n);
    for (int c : C) if (ponuka.count(c) == 0) return NIE; // rating cloveka c nik nema

    intervalac T(MAXR);
    for (int m=0; m<M; ++m) {
        int tz, tk; tie(tz, tk) = interval_hrany(m);
        if (tk < tz) continue;
        T.vloz_interval(m, tz, tk+1);
    }
    union_find UF(N);
    if (T.prejdi(UF)) return ANO; else return NIE;
}

int main() {
    int TC; cin >> TC; while (TC--) cout << vyries() << endl;
}
```

TRIDSIATY DEVIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek, Paulína Smolárová, Truc Lam Bui
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2024