P-III-4 Bandasky

Nejmenší počet bandasek, pro který řešení vždy existuje, najdeme tak, že najdeme situaci s nejvíce bandaskami, pro kterou ještě řešení neexistuje, a jejich počet zvětšíme o 1. Chceme tedy do stromu rozmístit co nejvíce bandasek tak, aby Jafar neuměl žádnou dostat do oázy j.

Představíme si, že vrchol j je kořenem našeho stromu a všechny dálnice jsou orientované směrem k j. Snadno nahlédnu, že Jafarovi se nikdy nevyplatí žádnou bandasku posílat po dálnici opačným směrem, tedy pryč od j. Pokud bychom z toho směru následně nikdy nedostali bandasku zpět, zbytečně jsme přišli o bandasku. A pokud jsme někdy v budoucnu nějakou dostali, mohli jsme vynechat dvě akce – tu, kde posíláme tuto bandasku tam, a tu, kde posíláme později bandasku zpět – a měli bychom stejně dobré řešení plus ušetřené dvě bandasky.

Kdykoliv, když máme v kterékoli oáze (alespoň) dvě bandasky, proto nic nezkazíme, když jednu z nich pošleme o krok blíž ke j – nic jiného s nimi už stejně neumíme dělat.

Dále si můžeme všimnout, že každý krok směrem pryč od cíle nám zdvojnásobuje potřebný počet bandasek. Mějme například cestu $1-2-3-4$. Máme-li v oáze 1 dvě bandasky, umíme jednu z nich dostat do oázy 2. Máme-li v oáze 1 čtyři bandasky, umíme udělat předchozí akci dvakrát, čímž dostaneme v oáze 2 dvě bandasky a tedy získáme možnost jednu z nich poslat do oázy 3. A podobně o krok dále: z osmi bandasek v oáze 1 budou čtyři v oáze 2, poté dvě v oáze 3 a z nich jedna v oáze 4.

Toto umíme popsat i formálněji. Podívejme se na situaci na konci a zaměřme se na libovolnou bandasku x, která stále existuje v nějakém vrcholu v. Nyní se podívejme na situaci na začátku a uvažme všechny bandasky, které jsme zapojili do toho, aby se bandaska x dostala do svého cíle. Pro každou z těchto bandasek (včetně x samotné) se podívejme na její vzdálenost d od vrcholu v a přiřaďme jí skóre $1/2^d$. Nyní tvrdíme, že součet skóre všech těchto bandasek je vždy přesně roven jedné.

Důkaz uděláme snadno matematickou indukcí dle počtu bandasek: Pokud je bandaska sama, je již v cíli a má skóre $1/2^0 = 1/1 = 1$. No a pokud kdykoliv máme dvě bandasky v libovolné vzdálenosti $d+1$ a vyrobíme z nich jednu bandasku ve vzdálenosti d, celkový součet skóre se tím nezmění.

Obecnější otázka

Zamysleme se nyní nad trochu obecnější otázkou: Mějme zakořeněný strom s kořenem v, který má děti $y_1, \ldots, y_k$. Kolik nejvíc bandasek můžeme v tomto stromu rozmístit tak, aby jich na konci v kořeni skončilo nanejvýš z?

Pokud v již nemá žádné děti ($k = 0$), odpověď je zjevně z. Nechť tedy v má $k > 0$ dětí. Pro každý z podstromů s kořeny $y_1, \ldots, y_k$ se podívejme na jeho hloubku – tedy na maximální vzdálenost mezi v a listem v tomto podstromu. Bez újmy na

obecnosti, nechť y_1 je vrchol s nejhlubším podstromem a nechť ℓ je jeden konkrétní nejhlubší list v tomto podstromu, ve vzdálenosti d od vrcholu v. Potom tvrdíme následující: existuje optimální rozmístění bandasek, při kterém ve v skončí právě z bandasek a všechny tam přijdou z vrcholu y_1.

Důkaz je zjevný: Vezměme libovolnou bandasku x, která skončila v kořeni v. Víme, že bandasky, ze kterých vznikla, měly dohromady skóre 1, a jelikož největší možná vzdálenost od kořene je d, nejmenší možné skóre jedné bandasky na začátku je $1/2^d$. Počet bandasek, z nichž bandaska x vznikla, je tedy nejvýše 2^d, a tedy jednou z optimálních možností je ta, kde bandaska v kořeni vznikla z 2^d bandasek, které všechny začínaly v listu ℓ.

Algoritmus

Z právě prokázaného tvrzení tedy vyplývá, že optimální rozmístění bandasek, které hledáme v předchozí části, umíme sestrojit následovně: Pokud v (kořen stromu) má děti, nechť y_1 je jeho dítě s nejhlubším podstromem. Potom:

- Samotný kořen bude na začátku prázdný.
- V podstromu s kořenem y_1 chceme bandasky rozmístit tak, aby jich do y_1 přišlo nanejvýš $2z + 1$, tj. největší počet bandasek, pro který jich pak odtud do kořene umíme poslat jen z.
- V každém z ostatních podstromů chceme bandasky rozmístit tak, aby do jeho kořene y_i přišla nanejvýš jedna.

Na každý z podstromů se proto rekurzivně zavoláme s příslušnou otázkou.

Pokud si na začátku jednou předpočítáme hloubky všech podstromů, umíme pak při tomto algoritmu každý vrchol zpracovat v čase přímo úměrném jeho stupni, a protože součet stupňů vrcholů ve stromu je $2n - 2$ (každá hrana je započtena ve dvou stupních vrcholů), má náš algoritmus celkově lineární časovou složitost.

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

const long long MOD = 1000000007;

int N, J;
vector< vector<int> > strom;
vector<int> rodic, hloubka, nejhlubsi_syn;
long long odpoved;

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

void vypln(int kde, long long kolik) {
    if (hloubka[kde] == 0) {
        // jsme v listu
        odpoved = (odpoved + kolik) % MOD;
    } else {
        // tlačíme dále
        vypln( nejhlubsi_syn[kde], (2*kolik+1)%MOD );
        for (int kam : strom[kde]) {
            if (kam == rodic[kde]) continue;
            if (kam == nejhlubsi_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);
    hloubka.resize(N, 0);
    nejhlubsi_syn.resize(N, -1);
    dfs(J);
    odpoved = 0;
    vypln(J, 0);
    cout << ((1+odpoved)%MOD) << endl;
}
```

P-III-5 Pračka

Pro $k = 0$ má pro každé $i < j$ platit $a_j - a_i \leq 0(j - i) = 0$, čili $a_i \geq a_j$. Jinými slovy, chceme vyrobit nerostoucí posloupnost.

Když si zvolíme, které pozice změnit, ty nezměněné musí tvořit nerostoucí podposloupnost původní posloupnosti. A naopak, když si zvolíme libovolnou nerostoucí podposloupnost, vždy umíme změnit všechny ostatní prvky tak, aby celý výsledek byl nerostoucí. Řešením je tedy najít jednu nejdelší nerostoucí podposloupnost a změnit zbytek.

Oprava posloupnosti na nerostoucí

Představme si, že jsme našli jednu konkrétní nejdelší nerostoucí podposloupnost. Všechny prvky, které do ní nepatří, přelepíme nálepkami. Na tyto nálepky chceme nyní napsat nové hodnoty tak, aby celá nová posloupnost byla nerostoucí.

Pokud posloupnost začíná nálepkou, napíšeme na ni největší hodnotu (tedy např. první nepřelepenou, nebo klidně rovnou 10^9), tím určitě nic nezkazíme. No a pak už stačí přejít zbytek posloupnosti zleva doprava a na každou další nálepku, kterou potkáme, napsat stejné číslo jako je těsně před ní. Výsledkem tohoto procesu je nerostoucí posloupnost a počet změn, které jsme udělali, je zjevně optimální.

Nalezení nejdelší nerostoucí podposloupnosti

Toto umíme udělat v čase $\mathcal{O}(n \log n)$ například následovně. Představme si, že jsme již zpracovali nějaký kus vstupní posloupnosti, například

$$(10, 17, 12, 14, 11, 16, 14, 1, 12, 15).$$

Z této posloupnosti umíme vybrat mnoho různých dvouprvkových nerostoucích posloupností, například. $(10, 5)$, $(16, 16)$, nebo $(16, 14)$.

Nyní nám na vstupu přijde nový prvek a_x. Představme si, že chceme vyrobit tříprvkovou nerostoucí podposloupnost, která končí tímto a_x. Toto musíme udělat tak, že vezmeme některou dvouprvkovou nerostoucí podposloupnost, kterou jsme měli v již zpracovaných datech, a na její konec přidáme a_x. No a která ze všech možných dvouprvkových podposloupností je k tomu nejvhodnější? Je zjevné, že pokud je například posloupnost $(10, 5, a_x)$ nerostoucí, tak i posloupnost $(16, 14, a_x)$ bude nerostoucí, neboť pokud $a_x \leq 5$, tak tím spíše platí i $a_x \leq 14$. Jinými slovy, nejlepší je ta posloupnost, která končí největším možným číslem. Každé a_x, které by pasovalo za nějakou jinou podposloupnost, bude pasovat i za tuto.

Budeme si tedy udržovat hodnoty c_j s následujícím významem: když se podíváme na všechny možné nerostoucí j-prvkové podposloupnosti v již zpracovaných datech a vezmeme poslední prvek každé z nich, největší z takto získaných hodnot bude právě c_j. Speciálně budeme mít $c_0 = \infty$ (za posloupnost délky 0 lze přidat cokoliv) a $c_j = -\infty$ pokud ještě ve zpracovaných datech žádná nerostoucí j-prvková podposloupnost neexistuje. Například, pokud jsme již zpracovali $(10, 17, 12, 14, 11, 16, 14, 1, 12, 15)$, tak budeme mít $c_1 = 17$, $c_2 = 16$, $c_3 = 15$, $c_4 = 12$ a od c_5 dále budou všechny mít hodnotu $-\infty$. Všimněte si, že různé c_i budou obecně

odpovídat různým posloupnostem. Např. zde c_3 odpovídá posloupnosti $(17, 16, 15)$, zatímco c_4 je konec posloupnosti $(17, 16, 14, 12)$.

Ke každému c_i si můžeme navíc pamatovat i index d_i, na kterém tato hodnota ležela. V našem příkladu bychom (používajíce indexování od nuly) měli $d_1 = 1$, $d_2 = 5$, $d_3 = 9$ a $d_4 = 8$. Jiný příklad: pokud jsme již zpracovali $(7, 7, 7)$, máme $c_1 = c_2 = c_3 = 7$, $c_4 = -\infty$ a indexy $d_1 = 0$, $d_2 = 1$ a $d_3 = 2$.

Dále použijeme následující pozorování: hodnoty c_i jsou vždy uspořádány podle velikosti v *nerostoucím* pořadí. Například vždy platí $c_3 \geq c_4$, neboť když vezmu *nejlepší* nerostoucí podposloupnost délky 4 a odstraním z ní poslední prvek, tak dostanu *nějakou* (ne nutně nejlepší) nerostoucí podposloupnost délky 3. A jelikož původní posloupnost končila hodnotou c_4, tato z ní vyrobená končí hodnotou větší nebo rovnou c_4.

Představme si nyní, že přečteme ze vstupu následující prvek a_x a zajímá nás, jaká nejdelší nerostoucí podposloupnost končí tímto prvkem. Abychom to zjistili, chceme najít největší i takové, že $c_i \geq a_x$. Potom je zjevné, že nejdelší nerostoucí podposloupnost končící právě přečteným a_x má délku $i + 1$. Jelikož víme, že hodnoty c jsou uspořádány podle velikosti, umíme hledané i nalézt v logaritmickém čase binárním vyhledáváním. Navíc si po nalezení správného i umíme pro pozici x zapamatovat, že nejlepší posloupnost končící zde vznikla prodloužením nejlepší posloupnosti končící na pozici d_i. Z takto zapamatovaných indexů umíme pak pro každou pozici jednu nejdelší na ní končící podposloupnost efektivně sestrojit.

Zbývá nám poslední krok: zjistit, jak se hodnoty c_i změní, když na konec zpracované posloupnosti přidáme právě přečtenou hodnotu a_x. Ukáže se, že změna je vždy jen minimální. Vzpomeňme si, že v předchozím kroku jsme našli největší i takové, že $c_i \geq a_x$. Nyní platí:

- Pro každé $j \leq i$ již i bez a_x existuje j-prvková nerostoucí podposloupnost končící prvkem větším nebo rovným a_x. Hodnoty c_1 až c_i se tedy nezmění.
- Nemáme žádný způsob, jak vyrobit nerostoucí podposloupnost délky $i+2$ a více, která by končila právě přečteným a_x. Ani hodnoty od c_{i+2} dále se tedy nezmění.
- Tím pádem jediné, co se změní, je hodnota c_{i+1}. Dosavadní c_{i+1} bylo ostře menší než a_x, nyní máme zjevně $c_{i+1} = a_x$.

Například pokud bychom pokračovali ve výše uvedeném příkladu tím, že přečteme ze vstupu jako další prvek posloupnosti hodnotu $a_x = 13$, změnila by se hodnota c_4 z 12 na 13. Pokud bychom místo toho přečetli $a_x = 12$, zůstala by $c_4 = 12$ a změnila by se c_5 z $-\infty$ na 12.

Po přečtení a zpracování n prvků budeme mít nanejvýš n-prvkovou nejdelší nerostoucí podposloupnost, proto je v libovolném okamžiku jen $\mathcal{O}(n)$ prvků posloupnosti c, které mají konečnou velikost, a tedy binární vyhledávání v nich běží v době $\mathcal{O}(\log n)$. Každý prvek ze vstupu tedy přečteme a zpracujeme v logaritmickém čase, a tím pádem je celková časová složitost tohoto řešení $\mathcal{O}(n \log n)$.

Obecné řešení

Podmínku, že pro všechny $i < j$ má platit $(a_j - a_i) \leq k(j - i)$, což můžeme upravit do ekvivalentní podoby: $(a_i - k \cdot i) \geq (a_j - k \cdot j)$.

Uvažujme nyní posloupnost $b_i = a_i - k \cdot i$. Tato posloupnost má pro všechny $i < j$ splňovat $b_i \geq b_j$, čili má být nerostoucí. To je ale přesně úkol, který už umíme řešit. Stačí tedy použít na posloupnost b výše popsané řešení, a pak z opravené posloupnosti b dopočítat opravenou posloupnost a.

Zbývá jediný technický detail: Zadání vyžaduje, aby všechny hodnoty opravené posloupnosti ležely v rozsahu od 1 do 10^9. K tomu stačí, abychom po výše popsané úpravě (provedené v neohraničených celých číslech, resp. v 64-bitových proměnných) všechny nové hodnoty, které vyšly větší než 10^9, změnili na rovné 10^9.

Hodnoty po opravě výše popsaným způsobem jsou zjevně nadále všechny kladné. Hodnoty, které neměníme, všechny zůstávají v povoleném rozsahu, takže nad 10^9 mohou vyběhnout jen doplňované nové hodnoty. No a rozborem případů snadno ověříme, že tím, že novým hodnotám nedovolíme překročit 10^9, nemohou vzniknout žádné špatné dvojice indexů $i < j$.

```
#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" : " ");
}
```

P-III-6 Šachisti

Ratingy tréninkem jen rostou, proto nikomu nikdy nesmíme zvýšit rating nad jeho požadovaný. Speciálně tedy platí, že pokud něčí počáteční rating r_i je ostře větší než jeho požadovaný rating c_i, řešení neexistuje. Toto můžeme na začátku zkontrolovat a poté ve zbytku řešení předpokládat, že pro každého šachistu platí $r_i \leq c_i$.

Pokud má člověk x na konci mít rating y, musí se k němu postupně dostat nějakou cestou od někoho, kdo má rating přesně y na začátku. Přes jaké jiné šachisty může taková cesta vést? Jsou právě dvě podmínky:

- Zjevně na této cestě nesmí být nikdo, kdo má už nyní větší rating než y.
- Také na ní nesmí být nikdo, kdo má na konci mít menší rating než y.

Takovou cestu nazývejme *platnou cestou* pro člověka x. Tvrdíme, že celkové řešení existuje právě tehdy, pokud pro každého šachistu existuje platná cesta.

Důkaz: Platné cesty použijeme uspořádané podle ratingu y, od nejmenšího po největší. Pro každého člověka se podívejme nejprve na okamžik, kdy jdeme použít jeho cestu k tomu, aby se on dostal svůj požadovaný rating y. Doposud jsme používali jen cesty s ratingem $\leq y$, každý člověk má tedy buď svůj počáteční rating, nebo je jeho rating nejvýše y. Nevznikly nám tedy žádní noví lidé s ratingem větším než y, a tedy naši cestu opravdu i nyní můžeme použít.

No a teď si uvědomme, že jakmile začneme používat cesty s ratingem větším než je rating y tohoto člověka, tyto cesty nemohou vést přes něj (již na začátku pro ně nesplňoval druhou podmínku), a tedy jeho rating již zůstane y až do konce.

Lehčí speciální typy grafů

Na úplném grafu je řešení našeho úkolu snadné: Stačí, aby každý z cílových ratingů c_i existoval v sadě počátečních ratingů r_i. Jelikož se každá dvojice přátelí, tak jako každou platnou cestu můžeme vzít přímou hranu.

Na hvězdě všechny cesty vedou přes střed a jen ten musíme kontrolovat.

Na dlouhé cestě si snadno uvědomíme, že když chceme dostat rating y šachistovi x, stačí uvažovat dvě možnosti: buď ho tam dostaneme od *nejbližšího* šachisty s počátečním ratingem y nalevo od x, nebo od nejbližšího takového napravo. Stačí tedy pro tyto dvě cesty zkontrolovat, zda jsou platné. Toto ověření umíme převést na otázky na minimum a maximum souvislého úseku v posloupnosti, a na takové otázky umíme efektivně odpovědět např. tak, že si nad posloupností postavíme intervalový strom.

Na malých grafech (ať už stromech nebo obecných) si můžeme dovolit z každého člověka spustit jedno prohledávání grafu, a tím ověřit, zda do něj existuje nějaká platná cesta. Stačilo by i prohledat celý graf jednou pro každou hodnotu cílového ratingu, ale asymptotickou časovou složitost nejhoršího případu nám tato optimalizace nezmění.

Velké stromy

Implementováním řešení pro výše popsané třídy grafů se dalo dohromady získat 8 bodů (z toho 6 za řešení pro malé grafy). Zbývají poslední dva body. V této části si ukážeme řešení, které získá ten devátý tím, že vyřeší sadu 8: velké stromy, ve kterých jsou počáteční ratingy r_i navzájem různé.

Pro každého šachistu x_1 existuje v celém stromě nanejvýš jeden šachista x_2, který má na začátku jeho požadovaný rating. Tím je určena celá cesta pro x_1, zůstává jen zkontrolovat, zda je platná.

Cestu z x_1 do x_2 si vždy umíme rozdělit na dvě (možná prázdné) cesty x_1x_3 a x_2x_3 tak, aby obě šli ve stromě jen vzhůru. Vrchol x_3, kde cestu rozdělíme, je nejbližším společným předkem x_1 a x_2. Stačí tedy, když pro každou cestu vzhůru stromem (od x_1 po x_3 a od x_2 po x_3) umíme říci, jaký největší začáteční a jaký nejmenší požadovaný rating na ní leží. Na toto si umíme předpočítat v čase $\mathcal{O}(n \log n)$ užitečné údaje následovně: pro každý vrchol x stromu a každé i (do $i = \log_2 n$) si spočítáme toto maximum a minimum pro cestu, která jde z x nahoru a má délku 2^i. Podobné předpočtené údaje umíme využít i pro efektivní nalezení nejbližšího společného předka. Více detailů o tomto algoritmu naleznete na *https://www.ksp.sk/kucharka/lca/* nebo *https://mj.ucw.cz/vyuka/ga/9-decomp.pdf*.

Každou cestu takto umíme celou zkontrolovat v čase $\mathcal{O}(\log n)$. Toto řešení má tedy časovou i paměťovou složitost $\mathcal{O}(n \log n)$.

Jiné efektivní řešení lze udělat pomocí tzv. heavy-light dekompozice stromu (*https://en.wikipedia.org/wiki/Heavy-light_decomposition*). Základní myšlenka je taková, že je-li strom košatý a mělký, tak jsme spokojeni, neboť všechny cesty jsou krátké a tedy můžeme na otázky odpovídat hrubou silou. Vadí nám jen stromy, které se málo větví a jsou hluboké. Při výše zmíněné dekompozici se ukáže, že umíme libovolném stromě najít malý počet dlouhých cest tak, že celý zbytek zůstane košatý. Pro každou cestu zvlášť pak použijeme výše popsané řešení pro cestu, a zbytek stromu si už můžeme dovolit zpracovat hrubou silou.

Takové řešení každého šachistu zkontroluje v čase $\mathcal{O}(\log^2 n)$, a tedy jeho celková časová složitost je $\mathcal{O}(n \log^2 n)$. Paměťová složitost je pouze $\mathcal{O}(n)$.

Velké obecné grafy

Zbývá nám vymyslet, jak získat poslední bod za tuto úlohu.

Vezměme si konkrétní požadovaný rating y. Teď si představme, že jsme smazali vrcholy, které nemohou být na cestě pro tento rating, tj. buď začínají s ratingem větším než y, nebo jejich požadovaný výsledný rating je menší než y. Zůstanou nám nějaké komponenty souvislosti. Kdy existují cesty pro šachisty, kteří chtějí tento rating? Zjevně právě tehdy, když pro každou komponentu platí: „pokud obsahuje někoho s požadovaným ratingem y, musí obsahovat i někoho se začátečním ratingem y“.

Podívejme se nyní na následující rating $y + 1$. Jak se změní náš graf? Začnou existovat vrcholy, které mají počáteční rating přesně $y + 1$ a přestanou existovat vrcholy, které mají požadovaný rating přesně y. Mělo by být zjevné, že když budeme postupně iterovat přes všechny ratingy, tak každý vrchol jen jednou začne a jednou přestane existovat.

Na ratingy, přes které postupně iterujeme, se můžeme dívat jako na čas. Pro každou hranu našeho grafu si umíme spočítat interval časů, po které existuje: jsou to časy, kdy najednou existují oba její koncové vrcholy.

Samotný průběh kontroly cesty

Na začátku si předzpracujeme vrcholy, abychom pro libovolné y uměli efektivně najít jak vrcholy s $r_i = y$, tak vrcholy s $c_i = y$. Představme si, že již máme sestrojeny výše popsané komponenty souvislosti pro nějaké konkrétní y – tedy pro každý vrchol

dokážeme efektivně říci ID komponenty, do které patří.

Jak ověříme, zda pro toto y existují všechny potřebné cesty? Stačí nejprve do prázdné množiny pro každý vrchol s $c_i = y$ vložit ID jeho komponenty, pak pro každý vrchol s $r_i = y$ z té množiny odstranit ID jeho komponenty (pokud tam je), a na závěr zkontrolovat, zda množina zůstala prázdná.

Za celý běh řešení bude každý vrchol jednou takto vložen a (nanejvýš) jednou odstraněn, takže dohromady bude čas strávený těmito kontrolami zanedbatelný oproti zbytku řešení.

Odmocninový trik

Pro každý přechod z času = ratingu y na $y+1$ se můžeme podívat na to, kolika hranám se změní stav. Počínaje od času 1 si nyní můžeme časovou osu nasekat na kusy následovně: postupně zvyšujeme konec aktuálního úseku a počítáme si změny stavu hran, které jsme viděli, dokud nepřijdeme na přechod, včetně kterého by již tento počet překročil $\sqrt{m}$. Tam ukončíme aktuální kus a od následujícího ratingu začneme další. Takto dostaneme $\mathcal{O}(\sqrt{m})$ kusů a uvnitř každého se děje nanejvýš $\sqrt{m}$ změn stavu hrany. Na některých konkrétních přechodech, které skončily na hranici mezi úseky, se může dít takových změn i řádově více. Tyto změny ale nikdy nebudeme zpracovávat!

Pro každý interval časů nyní uděláme následující: Sestrojíme si množinu hran, které existují během celého toho intervalu, a druhou malou množinu hran, které během něj mění stav. Jednou uděláme prohledávání (nebo Union-Find) na první množině hran a najdeme si komponenty souvislosti, které jim odpovídají. Potom zvlášť zpracujeme každý relevantní čas (tj. takový, který odpovídá někoho požadovanému ratingu) v daném intervalu.

Zpracování konkrétního času y tedy vypadá následovně:

- Projdeme $\mathcal{O}(\sqrt{m})$ hran, které mění stav během daného intervalu, a zjistíme, které existují v době y.
- Přidáme tyto hrany do grafu a v čase přibližně přímo úměrném jejich počtu určíme nové komponenty.
- Poté zkontrolujeme všechny vrcholy, které chtějí skončit na tomto ratingu y.

Takové řešení bude mít asymptotickou časovou složitost zhruba $\mathcal{O}(m\sqrt{m})$; přesná časová složitost závisí na konkrétní zvolené implementaci sestrojení komponent souvislosti a použitých datových strukturách.

Efektivnější řešení

Existují i řešení, která pro obecné grafy mají časovou složitost $\mathcal{O}(m \log n)$. Chcete-li se nad nějakým zamyslet, pro inspiraci uvedeme např. odkaz na link/cut stromy: *https://en.wikipedia.org/wiki/Link/cut_tree*. My si ukážeme myšlenkově i implementačně jednodušší řešení, které bude mít jeden logaritmus navíc – poběží tedy v čase $\mathcal{O}(m \log^2 n)$.

Představme si intervalový strom nad časovou osou. Pro každou hranu grafu si můžeme najít interval časů, po které existuje, a pak tento interval vložit do

tohoto intervalového stromu. Tím dostaneme pro každou hranu $\mathcal{O}(\log n)$ vrcholů intervalového stromu, které dohromady odpovídají jejímu celému intervalu existence. V každém z těchto vrcholů si tuto hranu poznamenáme.

Nyní prohledáme do hloubky tento intervalový strom. Vždy, když vejdeme do vrcholu, tak v něm zapamatované hrany začnou existovat, a vždy, když vrchol opustíme, tak existovat přestanou. Zjevně tedy vždy, když přijdeme do listu (t.j. vrcholu představujícího jeden konkrétní čas y), budeme mít sestrojenou správnou množinu hran: právě ty hrany, které existují v této době y. Můžeme tedy nyní zkontrolovat šachisty, kteří chtějí skončit s ratingem y.

Jak si během tohoto procesu udržujeme komponenty, abychom tyto kontroly uměli dělat efektivně? Budeme opět dělat Union-Find, tentokrát specificky s heuristikou „union by rank“ ale bez komprese cest, abychom při každé operaci Union udělali jen jednu změnu v paměti. Takový Union-Find má zaručenou časovou složitost $\mathcal{O}(\log n)$ na operaci. Vždy, když přidáváme hranu a děláme nějakou změnu v paměti pro Union-Find, uložíme si přepsané hodnoty na zásobník. No a vždy, když chceme nějakou hranu odstranit, z vrchu zásobníku vezmeme přepsané hodnoty a vrátíme je zpět.

Pro každý z $\mathcal{O}(n \log n)$ „kusů hran“ uděláme konstantní počet operací s datovou strukturou Union-Find, čímž dostáváme slibovanou časovou složitost.

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

const string ANO = "ANO", NE = "NE";

int N, M, MAXR;
vector<pair<int,int> > E;
vector<int> R, C;
map<int, vector<int> > nabizi, 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 kam) {
        while (int(undo_log.size()) > kam) {
            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 projdi(union_find &UF, int, int, int, int);
};

pair<int, int> interval_hrany(int m) {
    // uzavřený interval časů, v kterých existuje hrana m (možná prázdný)
    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::projdi(union_find &UF, int wx=0, int wy=0, int wlo=0, int whi=-1) {
    if (whi==-1) whi = 1<<L;
    // přidej hrany, které jsou v aktualním vrcholu
    int aktualni_stav = UF.undo_log.size();
    for (int m : data[wx][wy]) UF.spoj( E[m].first, E[m].second );
    // jsi-li v listu, zpracuj ho, jinak se rekurzivně zavolej
    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 : nabizi[rating]) komponenty.erase( UF.sef(x) );
        return_value = komponenty.empty();
    } else {
        return_value = projdi(UF, wx+1, 2*wy, wlo, (wlo+whi)/2);
        if (return_value) return_value = projdi(UF, wx+1, 2*wy+1, (wlo+whi)/2, whi);
    }
    // při odchodu odeber hrany
    UF.undo(aktualni_stav);
    return return_value;
}

string vyres() {
    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 NE; // rating nemůže klesnout
    nabizi.clear(); potrebuje.clear();
    for (int n=0; n<N; ++n) nabizi[R[n]].push_back(n);
    for (int n=0; n<N; ++n) potrebuje[C[n]].push_back(n);
    for (int c : C)
```

```
        if (nabizi.count(c) == 0) return NE; // rating člověka c nikdo nemá

    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.projdi(UF)) return ANO; else return NE;
}

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