A-II-1 Investičný guru

Táto úloha je riešiteľná *pažravo*: vieme postupne prechádzať vstup zľava doprava a postupne farbiť jednotlivé prvky podľa jednoduchých pravidiel. Najskôr si popíšeme toto riešenie a potom si zdôvodníme jeho správnosť.
Pre každú už použitú farbu si budeme pamätať posledné číslo, ktoré dostalo túto farbu. Takto vieme pri spracovaní ďalšieho prvku ľahko povedať, ktorá farba sa naň dá použiť a ktorá nie.
Postupne zľava doprava teraz pre každý prvok a_i postupnosti spravíme nasledovné:

- Ak existujú farby, ktoré sa naň dajú použiť, použijeme tú z nich, ktorej aktuálne číslo je najväčšie (a teda najbližšie ku a_i).
- Ak nie, zoberieme novú farbu a tou ofarbíme a_i.

Tento algoritmus zjavne vytvorí nejaké korektné ofarbenie, potrebujeme ale dokázať, že vždy použije najmenší možný počet farieb.
Intuícia za správnosťou nášho algoritmu je v tom, že ak máme na výber viac farieb, vždy vyberieme tú, ktorou „stratíme najmenej možností“. Napr. ak máme štyri použité farby, ku ktorým si práve pamätáme čísla 100, 200, 300 a 400 a príde nám ďalší prvok s hodnotou 350 tak:

- Ak by sme ho dali tou farbou, ktorú má 100, ostanú nám štyri farby s číslami 400, 350, 200 a **300**.
- Ak by sme ho dali tou farbou, ktorú má 300, ostanú nám štyri farby s číslami 400, 350, 200 a **100**.

No a je zjavné, že tá druhá možnosť je lepšia od prvej, lebo každé ofarbenie, ktoré sme vedeli spraviť v prvej situácii, vieme spraviť aj v druhej. (Prvok, ktorý mal v prvej situácii neskôr nasledovať za prvkom s hodnotou 300 zjavne môže namiesto toho nasledovať za prvkom s menšou hodnotou 100.) Môžeme len získať – ak by nám ako ďalší napr. prišiel prvok s hodnotou 150, v prvej možnosti by sme už museli použiť piatu farbu, v druhej ešte nie.

Formálny dôkaz správnosti bude trochu dlhší, lebo treba rozobrať o niečo viac možností. Spravíme ho sporom.

Nech teda existuje nejaké lepšie ofarbenie ako to naše. Spomedzi všetkých takých zoberme to, ktoré sa s naším zhoduje na čo najdlhšom prefixe. Nech a_i je prvý prvok, s ktorým chce lepšie riešenie spraviť niečo iné ako to naše.
Zjavne nemôže ísť o krok, v ktorom sme používali novú farbu, keďže v našom riešení je použitie novej farby vynútené a tým pádom by ľubovoľné iné riešenie, ktoré sa doteraz zhodovalo s naším, muselo spraviť to isté. Naše riešenie teda a_i ofarbilo niektorou z už skôr použitých farieb. Ostávajú teda dve možnosti: buď ho lepšie riešenie ofarbí inou z už použitých farieb, alebo sa lepšie riešenie rozhodne ho ofarbiť novou farbou. Ukážeme, že obe tieto možnosti vedú k sporu.

Prvá možnosť: nech naše riešenie ofarbilo a_i modrou, pričom predchádzajúce modré číslo bolo m. Lepšie riešenie ho namiesto toho ofarbí červenou, ktorej predchádzajúce číslo bolo c. Keďže modrú farbu pre a_i sme vyberali tak, aby m bolo najväčšie možné, vieme, že $c < m$. Potom ale dostávame práve situáciu, ktorú sme si ukázali na príklade vyššie: akékoľvek riešenie, ktoré sme vedeli spraviť po ofarbení a_i červenou vieme určite spraviť aj po ofarbení a_i modrou – len vo zvyšku nášho lepšieho riešenia vymeníme modrú a červenú.
To je ale spor s predpokladom, že zvolené lepšie riešenie sa s naším zhoduje v najdlhšom možnom prefixe – ukázali sme, že existuje rovnako dobré riešenie ktoré sa s naším zhoduje v aspoň jednom ďalšom kroku.

No a veľmi podobné je to aj v prípade, že sa lepšie riešenie rozhodlo začať používať novú farbu (povedzme žltú). Na túto situáciu sa môžeme dívať tak, ako keby táto farba už existovala a pamätali by sme si ku nej nulu. Opäť môžeme použiť ten istý argument ako vyššie: čokoľvek, čo vieme spraviť po ofarbení a_i na žlto, vieme spraviť aj po ofarbení a_i na modro, stačí vo zvyšku riešenia vymeniť modrú a žltú. Jediný rozdiel je v tom, že touto výmenou môžeme dostať ešte lepšie riešenie od toho nami pôvodne zvoleného, keďže je možné, že v tom upravenom žltú nikdy nebudeme musieť použiť. To ale stále nič nemení na tom, že opäť máme spor – našli sme iné riešenie, ktoré je lepšie od nášho, ale zhoduje sa s ním na dlhšom prefixe.

Implementácia

Ostáva zodpovedať ešte jednu otázku: ako vyššie popísané riešenie efektívne implementovať?
Na to si stačí všimnúť, že ak farby číslujeme v poradí, v akom sme ich použili, a v poli si pre každú farbu pamätáme posledné číslo, ktoré tú farbu dostalo, bude toto pole *vždy usporiadané zostupne*.
Rozmyslime si, prečo to tak je. Stačí si uvedomiť, že každý krok, ktorý v takom stave začne, musí v takom stave aj skončiť.
Ak sme nútení pridať novú farbu, je to preto, že aktuálny prvok je menší od všetkých práve zapamätaných hodnôt. Vtedy teda pridáme na koniec postupnosti novú najmenšiu hodnotu a všetko je ok.
V ostatných prípadoch platí, že ak aktuálne a_i dostane farbu j, tak vieme, že farba $j-1$ (ak existuje) má aktuálne číslo väčšie od a_i, keďže doteraz bolo pole usporiadané. To ale znamená, že keď zväčšíme číslo farby j na a_i, pole naďalej ostane usporiadané.

No a toto pozorovanie nám už dáva ľahký návod na efektívnu implementáciu: pre každé a_i nájdeme správnu farbu pomocou binárneho vyhľadávania. Takto dostávame riešenie s časovou zložitosťou $O(n \log n)$.
Respektíve, ak by sme chceli ísť veľmi do detailov, presnejší odhad je $O(n \log k)$, kde k je optimálny počet farieb – naše pole farieb nebude mať nikdy viac ako k prvkov.
(Aj bez vyššie spraveného pozorovania o usporiadanom poli sa dá táto úloha vyriešiť s rovnakou časovou zložitosťou, stačí si vyššie popísané záznamy udržiavať v usporiadanej množine. V tej potom vieme v logaritmickom čase aj nájsť správnu farbu pre ďalší prvok, aj odstrániť starý a vložiť nový záznam pre túto farbu.)

Listing programu (C++)

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
	int n;
	vector<int> a;

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

	// rovno si pamatame cele podpostupnosti
	vector<vector<int>> podpostupnosti;

	// umiestnime prvy prvok
	podpostupnosti.push_back({a[0]});

	for (int i = 1; i < n; ++i) {
		// binarne vyhladame spravnu podpostupnost
		int l = -1, r = podpostupnosti.size();
		while (r - l > 1) {
			int m = (r + l)/2;
			if (podpostupnosti[m].back() < a[i]) r = m;
			else l = m;
		}
		if (r == podpostupnosti.size()) {
			// musime zalozit novu
			podpostupnosti.push_back({a[i]});
		} else {
			podpostupnosti[r].push_back(a[i]);
		}
	}

	cout << podpostupnosti.size() << endl;
	for (auto && p : podpostupnosti) {
		for (auto && x : p) cout << x << "␣";
		cout << endl;
	}
}
```

Bonus na záver: Dilworthova veta

Táto úloha úzko súvisí s *Dilworthovou vetou.* To je matematické tvrdenie o čiastočných usporiadaniach.
Čiastočné usporiadanie si vieme predstaviť ako orientovaný graf. Vrcholy grafu sú prvky, ktoré porovnávame, a orientované hrany hovoria, ktorý prvok je „menší“ od ktorého.

Vyžadujeme pri tom dve vlastnosti: antisymetriu a tranzitivitu. Prvá vlastnosť znamená, že v porovnaniach nesmú byť žiadne spory: graf teda nesmie obsahovať žiadne cykly (vrátane slučiek). Druhá vlastnosť znamená, že vieme používať pravidlo „ak a je menšie ako b a to je menšie ako c, musí a byť menšie ako c“. Teda ak máme hrany $a \to b$ aj $b \to c$, musíme mať aj hranu $a \to c$.
Príklady čiastočného usporiadania:

1. Prvky, ktoré porovnávame, sú všetky možné podmnožiny množiny $\{0, 1, \ldots, n-1\}$. Hovoríme, že X je „menšie“ ako Y, ak X je vlastnou podmnožinou Y.
2. Prvky, ktoré porovnávame, sú rôzne veľké a rôzne tvarované matriošky od rôznych výrobcov. Matrioška a je menšia od b ak sa a do b fyzicky zmestí.
3. Prvky, ktoré porovnávame, sú rôzne objednávky na jazdu taxíkom: každá má dané miesto aj čas odkiaľ aj kam sa pôjde. Hovoríme, že objednávka a je menšia od objednávky b, ak je fyzicky možné jedným autom stihnúť najskôr a a potom b.
4. Prvky, ktoré porovnávame, sú prvky postupnosti, ktorú sme dostali na vstupe v našej súťažnej úlohe. Prvok na indexe i je „menší“ od prvku na indexe j ak $i < j$ a zároveň $a_i < a_j$. Inými slovami, „menší“ znamená, že vie byť použitý skôr v tej istej farbe.

Rozmyslite si, že v každom z týchto príkladov platí antisymetria (napr. ak sa matrioška a zmestí do b, určite sa b nezmestí do a) aj tranzitivita.
Tiež si rozmyslite, že v každom z príkladov môžu existovať dvojice prvkov, ktoré sú neporovnateľné. Napr. v prvom príklade množina $\{0, 1\}$ nie je ani menšia ani väčšia od množiny $\{0, 2, 3\}$. V treťom príklade ak máme dve objednávky v tom istom čase na opačných koncoch mesta, zjavne sa ani jednu nedá stihnúť pred tou druhou.

Pre čiastočné usporiadania môžeme ďalej definovať dva pojmy: *reťazec* a *antireťazec*.
Reťazec je ľubovoľná sada prvkov, ktoré sú všetky navzájom porovnateľné. (Z vlastností čiastočného usporiadania je zjavné, že každý reťazec má jednoznačne určené poradie svojich prvkov. V grafovej reprezentácii čiastočného usporiadania teda každý reťazec zodpovedá nejakej orientovanej ceste.)
Naopak, *antireťazec* je ľubovoľná sada prvkov, z ktorých žiadne dva nie sú porovnateľné.

Keď sa na prvky našej postupnosti dívame ako na vyššie popísaným spôsobom čiastočne usporiadanú množinu, vidíme, že naša súťažná úloha je vlastne „rozdeľ všetky prvky do čo najmenšieho počtu reťazcov“.
Toto je úloha, ktorú si môžeme klásť pre ľubovoľnú čiastočne usporiadanú množinu prvkov.

Môžeme si všimnúť, že vždy platí nasledovný dolný odhad: ak existuje antireťazec A veľkosti k, tak určite potrebujeme spraviť aspoň k rôznych reťazcov. Totiž žiadne dva prvky A nemôžu skončiť v tom istom reťazci.

Dilworthova veta hovorí, že tento dolný odhad je vždy tesný: najmenší počet reťazcov, do ktorých vieme rozdeliť všetky prvky, je vždy presne rovný veľkosti najväčšieho antireťazca. (Toto tvrdenie ponecháme bez dôkazu. Záujemcom oň odporúčame https://en.wikipedia.org/wiki/Dilworth%27s_theorem.)

V našej úlohe sú antireťazce zjavne práve všetky klesajúce podpostupnosti. To teda znamená, že z Dilworthovej vety vieme, že optimálne k je vždy rovné dĺžke najdlhšej vybranej klesajúcej podpostupnosti našej postupnosti.

Ak vás toto bonusové rozprávanie zaujalo, skúste si ešte rozmyslieť, čomu presne zodpovedá rozdelenie na čo najmenej reťazcov pre ostatné vyššie uvedené príklady čiastočných usporiadaní.

A-II-2 Električky

Zadaná úloha je spočítať v danom strome dvojice vrcholov, ktoré majú vzdialenosť presne k. Vo vzorových riešeniach ju vyriešime tak, že v danom strome spočítame cesty dĺžky k.

Všade v tomto vzorovom riešení pod pojmom *cesta* chápeme takú postupnosť vrcholov, v ktorej každé dva po sebe idúce vrcholy sú spojené hranou a žiaden vrchol sa neopakuje. Dĺžka cesty je počet hrán, ktorými prechádza. Dva vrcholy stromu sú teda vo vzdialenosti k vtedy a len vtedy, keď jediná cesta, ktorá ich spája, má dĺžku presne k.
Cesty líšiace sa len v orientácii budeme pri určovaní ich počtu považovať za totožné – teda cestu z x do y budeme považovať za tú istú cestu ako cestu z y do x.

Kvadratické riešenie

Pre každý vrchol stromu zvlášť si môžeme spočítať vzdialenosti z tohto vrcholu do všetkých ostatných. Toto vieme spraviť v lineárnom čase ľubovoľným prehľadávaním stromu, napr. do šírky (BFS) alebo do hĺbky (DFS). Takto získame počet vrcholov, ktoré sú vo vzdialenosti práve k od práve spracovaného vrcholu – resp. ekvivalentne počet ciest dĺžky k, ktoré v práve spracovanom vrchole začínajú. Za toto riešenie bolo 5 bodov.

Zakorenené stromy a cesty v nich

Cestu k lepším riešeniam začneme tým, že si náš strom v ľubovoľne zvolenom vrchole zakoreníme. To si môžeme predstaviť tak, že strom za tento vrchol zavesíme a všetky jeho hrany teraz smerujú dodola.
V zakorenenom strome má každý vrchol okrem koreňa práve jedného *rodiča* a môže mať ľubovoľne veľa *detí*. (Rodič je ten jeho sused, cez ktorého vedie cesta do koreňa. Deti sú jeho ostatní susedia. Rodič vrcholu je v strome nad ním, deti sú pod ním.)
Všimnime si, že keď sa po zakorenenom strome pohneme najskôr dodola (po jednej hrane) a potom dohora, pohyb dohora musí byť po hrane, ktorou sme práve prišli. Teda ak sme začali v nejakom vrchole x, zišli sme do niektorého jeho dieťaťa a následne sme sa museli vrátiť späť do x.

Pozrime sa teraz na ľubovoľnú cestu po zakorenenom strome. Tvrdíme, že keď ideme po tejto ceste, tak ideme v strome najskôr len hore a potom len dole.
Toto tvrdenie pomerne zjavne vyplýva z pozorovania, ktoré sme spravili vyššie. Totiž akonáhle spravíme prvý krok dodola tak už musia nasledovať len samé kroky dole – krok hore by totiž znamenal návrat do vrcholu, v ktorom sme už boli.

Pre každú cestu povieme, že jej *apex* je ten jej vrchol, ktorý je najbližšie ku koreňu stromu. Apex každej cesty je zjavne jednoznačne určený – je to ten vrchol, v ktorom meníme smer z „dohora“ na „dodola“.

Počty ciest vedúcich dodola

Pri nasledujúcich riešeniach nám bude užitočná informácia o počtoch ciest vedúcich v našom zakorenenom strome len smerom dole. V tejto časti si vysvetlíme, ako tieto počty efektívne spočítať.
Pre každý vrchol x a každé i (od 0 po k) nech $p(x, i)$ označuje počet ciest dĺžky i idúcich z vrcholu x dodola.
Ako určíme tieto hodnoty? Použijeme techniku *dynamického programovania* na stromoch. Hodnoty budeme postupne počítať smerom od listov ku koreňu. Ľahký spôsob implementácie je prehľadať strom do hĺbky, začínajúc v koreni. Pre každý vrchol x sa najskôr postupne rekurzívne zavoláme na všetky jeho deti (čím spočítame hodnoty p pre ne) a následne spracujeme samotný vrchol x.
Ako spočítať hodnoty $p(x, i)$ keď už poznáme tieto hodnoty pre všetky deti vrcholu x? Jednoducho: $p(x, 0) = 1$ a pre každé $i > 0$ platí, že $p(x, i)$ je súčtom hodnôt $p(y, i-1)$ cez všetky y ktoré sú deťmi vrcholu x. Totiž každá cesta dĺžky i dodola z vrcholu x začína krokom do nejakého dieťaťa y a odtiaľ pokračuje nejakou cestou dĺžky $i - 1$. Všetky tieto cesty sú zjavne navzájom rôzne, ich celkový počet je preto rovný súčtu ich počtov.
Celkový výpočet všetkých hodnôt $p(x, i)$ vyššie popísaným spôsobom trvá len $O(nk)$ času. Totiž pre každý vrchol si raz inicializujeme k hodnôt a potom pre každé jeho dieťa spravíme $O(k)$ krokov výpočtu. Keďže hrán v strome je len $n - 1$, dokopy existuje len $n - 1$ dvojíc (vrchol, jeho dieťa), a teda dokopy spracúvaním všetkých detí všetkých vrcholov strávime len $O(nk)$ krokov výpočtu.

Riešenie efektívne pre malé stupne vrcholov

Všetky cesty dĺžky k budeme v tomto aj nasledujúcom riešení počítať tak, že pre každý vrchol spočítame, koľko ciest dĺžky k v ňom má svoj apex. Takto zjavne každú cestu započítame práve raz.
Niektoré cesty, ktoré majú apex vo vrchole x, v ňom majú jeden zo svojich koncov. Pri pohľade z vrcholu x má každá takáto cesta postupne k krokov dodola. Už vieme, že počet takýchto ciest je $p(x, k)$.
Ako vyzerajú pri pohľade z vrcholu x ostatné cesty, ktoré tam majú apex? Vrchol x každú takúto cestu rozdelí na dve časti. Každá časť tvorí jednu cestu z vrcholu x dodola. Tieto dve časti musia začínať rôznymi hranami a musia mať súčet dĺžok práve k.
Ako ich spočítať? Pre každé dve deti $y_1 < y_2$ vrcholu x spočítame tie dvojice ciest, z ktorých jedna začína krokom do y_1 a druhá krokom do y_2. Koľko ich je? To vieme zistiť v čase $O(k)$. Vyskúšame všetky možnosti pre počet i ďalších hrán na prvej ceste. Ak má cesta z vrcholu y_1 dodola i hrán, cesta z vrcholu y_2 dodola musí mať $k - 2 - i$ hrán, aby sme dokopy dostali cestu dĺžky k. A koľkými spôsobmi vieme takúto cestu zostrojiť? Pre prvú cestu máme $p(y_1, i)$ možností, pre druhú máme $p(y_2, k - 2 - i)$ možností, a keďže každú prvú môžeme skombinovať s každou druhou, tieto počty medzi sebou vynásobíme.
Pre vrchol stupňa d takto postupne prejdeme $O(d^2)$ dvojíc jeho detí a každú spracujeme v čase $O(k)$. Dokopy teda tento konkrétny vrchol spracujeme v čase $O(d^2k)$.
Toto vo všeobecnosti nie je efektívne, ale v riešení za 8 bodov môžeme predpokladať, že $d \leq 5$, teda d je malá konštanta. Za tohto dodatočného predpokladu môžeme prehlásiť, že čas potrebný na spracovanie každého vrcholu je priamo úmerný k, a teda celková časová zložitosť je $O(nk)$.

Riešenie efektívne pre ľubovoľné stupne vrcholov

Od vyššie uvedeného riešenia už nie je ďaleko k takému, ktoré bude efektívne pre stromy ľubovoľného tvaru. Jediné, čo potrebujeme, je nahradiť postupné skúšanie všetkých dvojíc detí niečím efektívnejším.
Jeden spôsob, ako toto spraviť, vyzerá nasledovne. Pri spracúvaní vrcholu x budeme postupne prechádzať cez všetky jeho deti y. Pre každé y si položíme otázku, koľko existuje takých ciest s apexom y, ktoré majú dve časti, pričom jedna z nich ide cez y. Takto dokopy každú z týchto ciest započítame dvakrát.
Ako si odpovieme na túto otázku? Opäť, budeme postupne skúšať všetky možnosti pre počet i hrán, ktoré ešte vedú dodola z y. Akonáhle sme zvolili i, vieme, že máme $p(y, i)$ možností pre to, ako konkrétne vyzerá tento kus cesty. Koľko možností máme pre jej druhú časť?
Začínajúc vo vrchole x, druhá časť cesty má dĺžku $k - 1 - i$. Takýchto ciest je $p(x, k - 1 - i)$. My ale nechceme všetky z nich, len tie, ktoré nevedú cez ten istý vrchol y. Tu je ale ľahká pomoc: tých z nich, ktoré vedú cez y, je zjavne práve $p(y, k-2-i)$. Dobrých možností pre druhú časť cesty máme teda práve $p(x, k-1-i)-p(y, k-2-i)$.
Toto riešenie pre každý vrchol postupne spracuje každé jeho dieťa, pričom spracovanie každého dieťaťa trvá $O(k)$ krokov. Podobne ako vyššie teda môžeme usúdiť, že bez ohľadu na tvar stromu má toto riešenie časovú zložitosť $O(nk)$.

Listing programu (C++)

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

vector<vector<int>> al;
int k;
long long answer = 0;

vector<int> dfs (int index, int parent) {
        vector<int> ret(k + 1);
        for (int son : al[index]) if (son != parent) {
                auto got = dfs(son, index);
                answer += got[k - 1];
                for (int i = 0; i < k; ++i) {
                        answer += got[i] * ret[k - i - 1];
                }
                for (int i = 0; i < k; ++i) {
                        ret[i + 1] += got[i];
                }
        }
        // pridame aktualnu zastavku
        ret[0]++;
        return ret;
}
```

```
int main () {
        int n;
        cin >> n >> k;
        al = vector<vector<int>>(n);
        for (int i = 0; i < n-1; ++i) {
                int x, y;
                cin >> x >> y;
                x--; y--;
                al[x].push_back(y);
                al[y].push_back(x);
        }
        dfs(0, -1);
        cout << answer << endl;
}
```

Riešenie nezávisiace od k

Naša súťažná úloha má ešte jedno efektívne riešenie, a to s časovou zložitosťou $O(n \log n)$ bez ohľadu na to, ako veľké je k. Takéto riešenie nie je úplne striktne lepšie od toho vyššie uvedeného – to je použiteľné pre veľmi veľké n ak je k veľmi malé. V tejto súťažnej úlohe sme sa preto rozhodli udeľovať plný počet bodov za oba prístupy. Stále však platí, že toto nové riešenie je efektívne pre omnoho väčší rozsah možných k, takže z praktického hľadiska je od toho predchádzajúceho lepšie. Navyše technika, ktorú pri ňom používame, vedie k efektívnemu riešeniu aj omnoho ťažších problémov.
Toto riešenie je založené na pokročilej technike nazvanej *centroidová dekompozícia* stromu.
Keď zo stromu odstránime vrchol (spolu so všetkými hranami, ktoré doň vedú), vo všeobecnosti sa nám strom rozpadne na niekoľko menších komponentov.
V každom strome s n vrcholmi existuje aspoň jeden *centroid*: taký vrchol, ktorý keď odstránime, dostaneme samé malé komponenty – presnejšie, žiaden komponent nesmie mať viac ako $n/2$ vrcholov.
Centroid stromu vieme ľahko nájsť. Zakoreníme si ho a začneme v koreni. Ak pre každé dieťa koreňa platí, že v jeho podstrome je nanajvýš $n/2$ vrcholov, samotný koreň je centroidom. Ak to neplatí, tak musí existovať práve jedno dieťa d ktoré má vo svojom podstrome viac ako $n/2$ vrcholov. V takomto prípade sa presunieme do d a celú úvahu zopakujeme. Takto postupne ideme dole stromom až do okamihu, kým neprídeme do vrcholu c, pre ktorý po prvýkrát platí, že pod každým z jeho detí je nanajvýš $n/2$ vrcholov. Toto musí eventuálne nastať (najneskôr keď prídeme do listu, ktorý už žiadne deti nemá) a keď to nastane, je c hľadaným centroidom. (Aj ten komponent, ktorý obsahuje otca o vrcholu c, je dostatočne malý, keďže vieme, že pri pohľade z o bol nadpolovičný počet vrcholov stromu na opačnej strane hrany oc.)
No a keď vieme efektívne hľadať centroid, vieme mnohé problémy na stromoch riešiť technikou *rozdeľuj a panuj*. Ukážeme si, ako touto technikou spočítať všetky cesty dĺžky k v strome.
Naprogramujeme funkciu, ktorá na vstupe dostane strom a na výstupe vráti počet ciest dĺžky k v ňom. Táto funkcia bude vyzerať nasledovne:

1. Ak má strom len jeden vrchol, rovno odpovieme.
2. V čase $O(n)$ nájdeme centroid c nášho stromu.
3. V čase $O(n)$ spočítame všetky cesty dĺžky k, ktoré vedú cez vrchol c.
4. Odstránime c zo stromu, čím nám vznikne niekoľko komponentov – nových stromov nanajvýš polovičnej veľkosti.
5. Každá cesta, ktorú sme ešte nezapočítali, musí ležať celá v jednom komponente. Na každý komponent sa preto rekurzívne zavoláme a tým spočítame všetky cesty v ňom.

Takéto riešenie má celkovú časovú zložitosť $O(n \log n)$. Toto vieme zdôvodniť nasledovne: pre ľubovoľný vstup si výpočet na ňom vieme graficky znázorniť ako strom postupných rekurzívnych volaní, ktoré počas neho prebehnú. Tento strom má hĺbku $O(\log n)$, lebo pri každom rekurzívnom volaní sa aktuálna veľkosť stromu zmenší aspoň na polovicu. (Toto je ten dôvod, prečo používame práve centroid ako to miesto, kde strom rozdeliť.)
No a v každej hĺbke rekurzie dokopy strávime $O(n)$ času. To preto, že v každom konkrétnom volaní strávime čas priamo úmerný veľkosti práve spracúvaného stromu. No a to, čo robíme, je, že náš pôvodný strom len delíme

na menšie. V úplne prvom volaní na pôvodný strom raz spracujeme každý jeho vrchol. Keď odstránime jeho centroid a postupne sa zavoláme na jednotlivé komponenty, tak každý vrchol okrem centroidu prispeje k časovej zložitosti v práve jednom z týchto volaní – tom, do ktorého komponentu patrí. A tak ďalej.
Jediná netriviálna časť tohto riešenia je ako v čase $O(n)$ spočítať cesty dĺžky k vedúce cez centroid. Toto spravíme podobne ako v predchádzajúcom vzorovom riešení. Zakoreníme si náš strom v c. Pre každého syna x vrcholu c spustíme z neho prehľadávanie do hĺbky a počas neho si pre každú vzdialenosť od x pamätáme, koľko vrcholov v tejto vzdialenosti sme už videli. Tým spočítame všetky nenulové hodnoty $p(x, i)$ v čase priamo úmernom veľkosti tohto podstromu. Dokopy teda takto v čase $O(n)$ získame relevantné dáta pre každého syna vrcholu c a z nich už ľahko určíme hľadaný celkový počet ciest.

A-II-3 Čokoláda s hrozienkami

Začneme tým, že si rozmyslíme, že riešenie vždy existuje – a teda nikdy nebude treba podať správu, že čokoládu nejde rozdeliť.
Dokonca vždy existuje aj riešenie, ktoré vieme dosiahnuť iba obyčajným lámaním čokolády. (Toto navyše bude pravda aj o optimálnom riešení.)

Jeden možný postup, ako zostrojiť platné riešenie, vyzerá nasledovne.
Pozrime sa na stĺpce čokolády, v ktorých ležia nejaký hrozienka. Najľavejší takýto stĺpec necháme na pokoji a pre každý ďalší čokoládu zlomíme tesne naľavo od neho.
Keď táto fáza skončí, máme niekoľko obdĺžnikov čokolády. Na každom z nich zjavne sú nejaké hrozienka, keďže na každom z nich skončil presne jeden zo stĺpcov, ktoré obsahovali nejaké hrozienka. A navyše teda na každom obdĺžniku všetky hrozienka ležia v tom istom stĺpci.
Zvlášť pre každý z týchto obdĺžnikov teraz môžeme postupovať tak, že ho zoberieme, otočíme o 90 stupňov a zopakujeme preň vyššie uvedený postup. Tým ho zjavne rozdelíme na menšie obdĺžniky, z ktorých každý bude obsahovať práve jedno hrozienko.
(Za implementáciu tohto alebo ľubovoľného iného platného delenia čokolády ste mohli získať 3 body.)

Uvažujme teraz najväčší možný obdĺžnik našej čokolády, ktorý obsahuje práve jedno hrozienko – to veľké, ktoré chceme pre seba. (Ak je takýchto obdĺžnikov viac, vyberme si ľubovoľný jeden z nich.)
Optimálne riešenie našej úlohy nemá ako byť väčšie od tohto obdĺžnika. Ak by sme teda vedeli rozdeliť čokoládu tak, aby toto bol jeden z n obdĺžnikov, bude takéto delenie zjavne optimálne.
Pozrime sa najskôr na ľubovoľný zo štyroch okrajov nášho obdĺžnika. Pre tento okraj sú len dve možnosti: buď je zhodný s okrajom celej čokolády, alebo je tesne pri ňom zvonka políčko obsahujúce iné hrozienko. (Ak by totiž ani jedno z toho nebola pravda, vedeli by sme náš obdĺžnik v tomto smere ešte zväčšiť.)
Teraz ukážeme, že vždy vieme čokoládu rozdeliť (dokonca rozlámať) tak, že nám ostane práve náš zvolený optimálny obdĺžnik.
Budeme postupovať nasledovne: Ak náš obdĺžnik nesiaha až po ľavý okraj čokolády, rozlomíme čokoládu pozdĺž ľavého okraja obdĺžnika. To isté následne spravíme s pravým, horným a dolným okrajom. (Pri hornom a dolnom okraji už lámeme len aktuálny kus čokolády, zlom presne zodpovedá príslušnej strane nášho obdĺžnika.)
O každom z nanajvýš štyroch kusov čokolády, ktoré takto odlomíme od nášho, vieme, že obsahuje aspoň jedno hrozienko – to, kvôli ktorému náš kúsok v danom smere nesiahal ďalej. Na každý kus následne vieme použiť vyššie popísaný postup a rozlámať ho tak na obdĺžniky obsahujúce po jednom hrozienku.
Tým sme urobili významný krok k riešeniu pôvodnej úlohy: teraz už vieme, že nám stačí nájsť najväčší obdĺžnik, ktorý obsahuje práve naše hrozienko a žiadne iné.

Pomalšie spôsoby hľadania

V mriežke rozmerov $r \times s$ je rádovo r^2s^2 možností, ako vyznačiť obdĺžnik: máme rádovo rs možností pre jeho ľavý horný roh, rádovo rs možností pre pravý dolný roh, a tými už je jednoznačne určený. Pre každý takýto obdĺžnik vieme v čase priamo úmernom jeho obsahu, teda $O(rs)$, skontrolovať, či obsahuje práve jedno hrozienko – to naše. No a už samozrejme stačí vybrať najväčší vyhovujúci. Takto dostávame riešenie s časovou zložitosťou

$O(r^3s^3)$ – alebo $O(r^2s^2n)$ ak namiesto bitmapy čokolády pri kontrole každého obdĺžnika prejdeme celý zoznam hrozienok.
Pre lepšie riešenia sa na čokoládu pozrime ako na maticu A jednotiek (malé hrozienka) a núl (ostatné políčka). My na čokoláde hľadáme najväčší obdĺžnik, ktorý je tvorený samými nulami a navyše obsahuje veľké hrozienko. Pre maticu A si vieme v čase $O(rs)$ spočítať jej dvojrozmerné prefixové súčty: novú maticu S takú, že $S[i, j]$ je rovné súčtu všetkého v prvých i riadkoch a j stĺpcoch matice A. Pomocou S vieme zistiť súčet ľubovoľného obdĺžnika v matici A v konštantnom čase. To nám zlepší časovú zložitosť predchádzajúceho riešenia na $O(r^2s^2)$. Detaily o prefixových súčtoch nájdete tu: https://www.ksp.sk/kucharka/2d_prefixove_sumy/.
Ešte o rád lepšie riešenie vieme dostať nasledovne: Postupne vyskúšame všetky možnosti pre najľavejší stĺpec s_1, v ktorom leží hľadaný obdĺžnik. Teraz budeme postupne skúšať všetky možnosti pre najpravejší stĺpec s_2. Ako postupne zväčšujeme s_2, niektoré riadky postupne prestávajú byť použiteľné, lebo už obsahujú nejaké zlé hrozienko. Keď si v poli priebežne pamätáme, ktoré riadky sú momentálne použiteľné, tak pre každú dvojicu (s_1, s_2), pre ktorú veľké hrozienko leží v správnom rozsahu, vieme v $O(r)$ zistiť, koľko riadkov v okolí toho správneho ešte stále vieme použiť, a teda aké najväčšie riešenie existuje pre tieto konkrétne stĺpce. Toto riešenie má teda časovú zložitosť $O(rs^2)$.

Prvé efektívne riešenie

Pozrime sa na riadok s veľkým hrozienkom. Ak sú v ňom ešte nejaké iné hrozienka, najbližšie vľavo a vpravo nám udajú nové hranice odkiaľ a pokiaľ má zmysel skúšať obdĺžniky obsahujúce veľké hrozienko.
Pre každý stĺpec v tomto rozsahu si predpočítame dva údaje: či a kde je v ňom najbližšie hrozienko nad a pod riadkom, ktorý obsahuje naše veľké hrozienko. Tieto údaje nám povedia, najviac pokiaľ dohora a dodola môže v tomto stĺpci siahať nami hľadaný obdĺžnik.
Celé toto predpočítanie vieme spraviť v čase $O(rs)$, lebo len raz prejdeme každý stĺpec.
Teraz vieme spraviť to isté ako v predchádzajúcom riešení, ale skúšať budeme len (s_1, s_2) z rozsahu, ktorý sme našli v prvom odseku, a vďaka potom predpočítaným údajom vieme po každom pridaní nového stĺpca v konštantnom čase prepočítať, pokiaľ dohora a dodola aktuálne môže siahať náš obdĺžnik. Táto fáza riešenia teda bude bežať v čase $O(s^2)$ a dokopy tak dostávame časovú zložitosť $O(rs + s^2)$.

Listing programu (C++)

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

void nakrajaj_vodorovne(int r1, int c1, int r2, int c2, vector<int> hr) {
    // vypise jedno mozne rozkrajanie obdlznika [r1,r2) x [c1,c2)
    // ktory ma vsetky hrozienka v tom istom stlpci v riadkoch hr[i]
    sort( hr.begin(), hr.end() ); // slo by nahradit countsortom
    hr.push_back(r2);
    int top = r1;
    for (int i=0; i+1<int(hr.size()); ++i) {
        cout << top << " " << c1 << " " << (hr[i+1]-1) << " " << (c2-1) << endl;
        top = hr[i+1];
    }
}

void nakrajaj_zvisle(int r1, int c1, int r2, int c2, const vector<int> &hr, const vector<int> &hc) {
    // vypise jedno mozne rozkrajanie obdlznika [r1,r2) x [c1,c2)
    // ktory ma hrozienka v bodoch (hr[i], hc[i])
    map<int, vector<int> > stlpce_hrozienok;
    for (int i=0; i<int(hc.size()); ++i) {
        // preskocime hrozienka ktore lezia mimo nasho obdlznika
        if (!(r1 <= hr[i] && hr[i] < r2 && c1 <= hc[i] && hc[i] < c2)) continue;
        stlpce_hrozienok[hc[i]].push_back(hr[i]);
    }
    vector<int> suradnice;
    for (const auto &rec : stlpce_hrozienok) suradnice.push_back(rec.first);
    suradnice.push_back(c2);
    int left = c1;
    for (int i=0; i+1<int(suradnice.size()); ++i) {
        nakrajaj_vodorovne(r1, left, r2, suradnice[i+1], stlpce_hrozienok[suradnice[i]]);
        left = suradnice[i+1];
    }
}

int main() {
    int R, C, N;
    cin >> R >> C >> N;
    vector<int> hr(N), hc(N);
```

```
    for (int n=0; n<N; ++n) cin >> hr[n] >> hc[n];

    // zisti rozsah stlpcov v ktorych moze lezat nas obdlznik
    int cl = 1, cr = C+1;
    for (int n=1; n<N; ++n) if (hr[n] == hr[0]) {
        if (hc[n] < hc[0]) cl = max( cl, hc[n]+1 );
        if (hc[n] > hc[0]) cr = min( cr, hc[n] );
    }

    // pre kazdy z tychto stlpcov zistime rozsah riadkov v ktorom mozeme byt v nom
    vector<int> rt(C+1, 1), rb(C+1, R+1);
    for (int n=1; n<N; ++n) {
        int r = hr[n], c = hc[n];
        if (c < cl || c >= cr) continue;
        if (r < hr[0]) rt[c] = max( rt[c], r+1 );
        if (r > hr[0]) rb[c] = min( rb[c], r );
    }

    // vyskusame vsetky moznosti pre nas obdlznik a vyberieme najvacsiu z nich
    int br1 = -1, bc1 = -1, br2 = -1, bc2 = -1, barea = 0;
    for (int c1=cl; c1<cr; ++c1) {
        int top=1, bot=R+1;
        for (int c2=c1+1; c2<=cr; ++c2) {
            top = max(top, rt[c2-1]);
            bot = min(bot, rb[c2-1]);
            if (c2 <= hc[0]) continue; // este neobsahuje nase hrozienko
            int area = (bot-top)*(c2-c1);
            if (area > barea) {
                barea = area;
                br1 = top; bc1 = c1; br2 = bot; bc2 = c2;
            }
        }
    }

    // vypiseme nas obdlznik
    cout << barea << endl;
    cout << br1 << " " << bc1 << " " << (br2-1) << " " << (bc2-1) << endl;

    // nakrajame zvysok, ak treba: najskor vlavo a vpravo
    if (bc1 > 1) nakrajaj_zvisle(1, 1, R+1, bc1, hr, hc);
    if (bc2 < C+1) nakrajaj_zvisle(1, bc2, R+1, C+1, hr, hc);
    // a potom co ostalo hore a dole
    if (br1 > 1) nakrajaj_zvisle(1, bc1, br1, bc2, hr, hc);
    if (br2 < R+1) nakrajaj_zvisle(br2, bc1, R+1, bc2, hr, hc);
}
```

Druhé efektívne riešenie

Existujú aj riešenia, ktoré majú časovú zložitosť čisto $O(rs)$. Hlavná myšlienka jedného takéhoto riešenia: upravíme jeden „klasický“ algoritmus, ktorým vieme v čase $O(rs)$ nájsť najväčší obdĺžnik tvorený samými nulami. Takýto algoritmus vieme implementovať tak, že postupne skúšame zhora dole všetky možnosti pre to, v ktorom riadku tento obdĺžnik končí. Pre konkrétny riadok úlohu vyriešime tak, že si ku každému stĺpcu pamätáme, ako veľa nulami končí, a teda ako najvyšší obdĺžnik vieme v ňom mať. Potom si tieto stĺpce núl v lineárnom čase usporiadame od najvyššieho po najnižší a v tomto poradí ich „povoľujeme“. Vždy, keď povolíme stĺpec, tak vieme v konštantnom čase povedať, pokiaľ doľava a doprava siaha najväčší obdĺžnik, ktorý tento stĺpec obsahuje – môžeme použiť práve všetky skôr povolené stĺpce.
Existuje aj iná technika, pri ktorej použijeme prechod poľom výšok stĺpcov zľava doprava a zásobník, v ktorom si pamätáme, do akej výšky sa aktuálne dá ísť ako ďaleko doľava.
Kľúčové pozorovanie je teraz nasledovné: Obe verzie tohto algoritmu majú dôležitú spoločnú vlastnosť: najväčší prázdny obdĺžnik nájdu tak, že prezrú nejakú sadu kandidátov a z nich vyberú ten celkovo najväčší. No a táto sada kandidátov obsahuje, okrem iného, úplne všetky *nezväčšiteľné* obdĺžniky – teda také, ktoré už nevieme zväčšiť ani o riadok ani o stĺpec do žiadnej strany.
Ak sa teraz pozrieme na našu čokoládu tak, že veľké hrozienko je tiež nula (rovnako ako prázdne štvorčeky), tak bude platiť, že optimálny kúsok, ktorý si chceme nechať, zodpovedá nejakému nezväčšiteľnému obdĺžniku núl – presnejšie najväčšiemu z tých, ktoré obsahujú veľké hrozienko. No a keďže vyššie popísaný algoritmus postupne prezrie *všetky* nezväčšiteľné obdĺžniky, stačí pre každý z nich otestovať, či obsahuje veľké hrozienko, a vybrať najväčší z tých, pre ktoré je to pravda.

A-II-4 Hviezdne impérium, čokoláda a strom

Podúloha A: spočítanie čokolád

Keby sme takúto úlohu riešili sekvenčne, vyzeralo by to celé tak, že postupne prejdeme celú cestu a v pomocnej premennej p si počas toho počítame, koľko systémov s čokoládou sme už videli.
Ako takéto niečo vieme spraviť paralelne pomocou veštieb?
Hlavnou myšlienkou je, že každý systém si nechá vyveštiť hodnotu, ktorá bola v p pred jeho spracovaním, a potom svojmu nasledovníkovi na ceste pošle príslušne zmenenú hodnotu (t.j. p ak v tomto systéme čokoládu nepoznajú alebo $p + 1$ ak áno).
Každý systém si potom skontroluje, či mu od predchodcu na ceste prišla tá hodnota p, ktorú mal vyveštenú. Posledný systém navyše skontroluje, či hodnota p, ktorú už nemal komu ďalej poslať, je aspoň rovná $n/3$.
Toto riešenie potrebuje $k = \lceil \log_2 n \rceil + 1$ bitov: vyveštené číslo p neprekročí n, a okrem neho si ešte potrebujeme dať vyveštiť jeden bit, ktorý nám povie, ktorý z našich dvoch susedov je na ceste pred nami.

Podúloha B: test stromu

Jedno veľmi stručné riešenie tejto úlohy vyzerá nasledovne:

- Dáme si vyveštiť číslo od 0 po $n - 1$.
- Rozpošleme svoje vyveštené číslo všetkým susedom.
- Ak od aspoň dvoch susedov dostaneme číslo menšie alebo rovné od nášho, odpovieme NIE.
- Ak nik neodpovedal NIE, odpovieme ANO.

Prečo toto riešenie funguje?
Ak je Hviezdne impérium strom, tak existujú také veštby, pre ktoré náš algoritmus odpovie ANO: napríklad si môžeme ľubovoľne zvoliť jeden systém ako koreň stromu a potom každému systému vyveštiť jeho vzdialenosť od koreňa. V každom strome platí, že každý vrchol má nanajvýš jedného suseda, ktorý je bližšie ku koreňu ako on sám.
(Presnejšie, koreň stromu dostane vyveštené číslo 0 a následne mu každý jeho sused pošle číslo 1. Ľubovoľný iný vrchol, ktorý dostane vyveštené číslo $k > 0$, dostane od práve jedného suseda číslo $k - 1$ a od všetkých ostatných susedov číslo $k + 1$. Každý vrchol teda odpovie ANO.)
Naopak, ak Hviezdne impérium obsahuje nejakú kružnicu C, žiadna dobrá sada veštieb neexistuje. Pozrime sa totiž na to, čo sme vyveštili vrcholom na kružnici C. Nech x je ten z nich, ktorý dostal najväčšie vyveštené číslo. (Ak je takých viac, tak ľubovoľný z nich.) Potom x určite dostane od susedov aspoň dve čísla menšie alebo rovné od vlastného – oba vrcholy, s ktorými susedí na kružnici C, mu takéto čísla pošlú. Tento vrchol x teda odpovie NIE.
Toto riešenie potrebuje $k = \lceil \log_2 n \rceil$ vyveštených bitov.

TRIDSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2023