P-II-1 Investiční guru

Pro libovolnou posloupnost $a_1, \ldots, a_n$ označíme jako její *podposloupnost* jakoukoliv posloupnost $a_{i_1}, \ldots, a_{i_m}$, kde $1 \leq i_1 < \cdots < i_m \leq n$. Jinak řečeno, podposloupnost musí zachovat pořadí členů, ale může některé z nich přeskočit. V úloze máme za úkol rozdělit posloupnost na co nejmenší počet rostoucích podposloupností.

Zkusme nejprve vyřešit jednodušší úlohu: Na vstupu dostaneme posloupnost $a_1, \ldots, a_n$ a číslo k a chceme posloupnost rozdělit na nejvýše k rostoucích podposloupností. Efektivní řešení této úlohy lze potom využít pro zisk 6 bodů: Postupně jej zavoláme s $k = 1, 2, \ldots, 5$ a vybereme nejmenší k, pro které bude odpověď ano.

Hladový algoritmus

Tuto úlohu vyřešíme hladovým algoritmem. Budeme postupně procházet posloupnost $a_1, \ldots, a_n$ a přitom konstruovat k rostoucích podposloupností. Na začátku si vytvoříme k prázdných posloupností. V kroku i potom přidáme a_i na konec takové posloupnosti, která zatím končí na nějaké číslo menší než a_i. Pokud je takových více, vybereme tu z nich, která končí největším číslem. Pokud žádná taková není a máme ještě nějaké prázdné posloupnosti k dispozici, přidáme a_i do některé z nich, jinak skončíme a vypíšeme, že posloupnost na k rostoucích podposloupností rozdělit nešla.

Nejprve dokážeme, že tento algoritmus úlohu vyřeší správně. Pokud doběhne až do konce, tak zřejmě každá z k vytvářených podposloupností bude rostoucí, protože prvek a_i přidáváme vždy buď do prázdné posloupnosti, anebo na konec nějaké, jejíž předchozí poslední prvek byl menší než a_i. Zároveň každý člen původní posloupnosti bude v některé z podposloupností, a tedy skutečně najdeme způsob, jak původní posloupnost rozdělit na k rostoucích podposloupností.

Pokud algoritmus v nějakém kroku selže a řekne, že úloha nemá řešení, potřebujeme dokázat, že úloha skutečně nemá řešení. Pro spor předpokládejme, že existuje *lepší* rozdělení posloupnosti $a_1, \ldots, a_n$ na k rostoucích podposloupností, a ze všech možných vyberme to, které se nejdéle shoduje s rozdělením, které se snaží konstruovat náš algoritmus. Označme jej $(b_{1,1}, \ldots, b_{1,n_1})$, $(b_{2,1}, \ldots, b_{2,n_2})$, $\ldots$, $(b_{k,1}, \ldots, b_{k,n_k})$.

Nechť i je první krok, kdy se náš algoritmus odlišuje od hypotetického lepšího rozdělení. To znamená, že náš algoritmus chce umístit a_i do jiné podposloupnosti než hypotetické lepší řešení. Nemůže se stát, že by se náš algoritmus pokusil dát a_i na začátek prázdné posloupnosti, nebo by skončil, jelikož to udělá jen tehdy, když všechny ostatní posloupnosti končí na čísla větší než a_i, a tehdy by i lepší rozdělení muselo založit novou posloupnost, protože se s hladovým algoritmem doteď shodovalo.

Máme tedy čísla $1 \leq f, g \leq k$ a indexy $1 \leq u < n_f$ a $1 \leq v \leq n_g$ takové, že $a_i = b_{f,u+1}$, ale náš algoritmus umístí a_i do g-té podposloupnosti za prvek $b_{g,v}$. Z toho

plyne, že $a_i > b_{g,v} > b_{f,u}$: Obě čísla $b_{g,v}$ a $b_{f,u}$ jsou zřejmě menší než a_i a zároveň náš algoritmus ze všech takových vybral to největší, totiž $b_{g,v}$. Tudíž můžeme místo těchto dvou posloupností uvažovat posloupnosti $(b_{f,1}, \ldots, b_{f,u}, b_{g,v+1}, \ldots, b_{g,n_g})$ a $(b_{g,1}, \ldots, b_{g,v}, a_i = b_{f,u+1}, b_{f,u+2}, \ldots, b_{f,n_f})$. Ty jsou také rostoucí, protože $b_{f,u} < b_{g,v} < b_{g,v+1}$ a $b_{g,v} < a_i$, takže dostáváme rozdělení na k rostoucích posloupností, které se s naším algoritmem shoduje ještě o krok déle. To je spor, protože jsme předpokládali, že vybrané řešení se s našim shoduje co možná nejdéle. Tím jsme dokázali správnost našeho algoritmu.

Pro implementaci algoritmu je potřeba vyřešit jen jeden detail, totiž jak nalézt podposloupnost, která končí na největší menší prvek než a_i. Jedna možnost je udržovat si podposloupnosti v nějakém binárním vyhledávacím stromu seřazené podle posledního prvku. Ve skutečnosti si ale všimněme, že budeme-li udržovat podposloupnosti v poli v pořadí, v jakém jsme do nich přidali první prvek, bude toto pole vždy seřazené sestupně podle velikosti posledního prvku: Pokud přidáváme a_i do nové posloupnosti, tak víme, že všechny již existující posloupnosti končí na prvky větší než a_i. Pokud přidáváme a_i na konec některé posloupnosti, víme, že a_i je větší než její poslední prvek (a tedy i než poslední prvky všech následujících posloupností), ale zároveň jsou poslední prvky všech předchozích posloupností větší než a_i. Jinak řečeno nám vždy stačí použít obyčejné binární vyhledávání. Tím za přidání jednoho prvku zaplatíme $\mathcal{O}(\log k)$ času, a tedy celý algoritmus poběží v čase $\mathcal{O}(n \log k)$.

Binární vyhledávání odpovědi

Předpokládejme, že máme nějakou úlohu s nějakým parametrem k a k ní algoritmus, jenž umí rozhodnout, zda existuje řešení této úlohy. Dále předpokládejme, že existuje-li řešení pro hodnotu parametru k, pak existuje i pro všechny hodnoty $\ell \geq k$. A nakonec si ještě pro zjednodušení slibme, že víme, že parametr bude vždy celé číslo v intervalu $[a, b]$, přičemž pro a řešení nikdy neexistuje a pro b vždy existuje. Potom můžeme pomocí binárního vyhledávání na intervalu $[a, b]$ nalézt nejmenší k, pro které algoritmus odpoví ano. Jestliže algoritmus trvá $\mathcal{O}(T)$ času, potom jej spustíme $\mathcal{O}(\log(b-a))$ krát, a tedy celková složitost bude $\mathcal{O}(T \log(b-a))$. Této technice se obvykle anglicky říká *binary search by the answer*.

Naše úloha všechny tyto předpoklady jistě splňuje: Rozdělení na k posloupností je samo o sobě i rozdělení na libovolných $\ell \geq k$ posloupností: můžeme si představit, že ty zbylé posloupnosti jsou prázdné. Také nikdy nedokážeme najít rozdělení na 0 rostoucích posloupností a vždy dokážeme posloupnost rozdělit na n rostoucích posloupností (například každý prvek bude ve vlastní posloupnosti). To znamená, že získáváme řešení, které běží v čase $\mathcal{O}(n(\log n)^2)$, kde jeden logaritmus je za binární vyhledávání odpovědi a druhý za binární vyhledávání uvnitř algoritmu.

Vzorové řešení

Přestože je binární vyhledávání odpovědi užitečná technika (a obecně je často dobrý nápad se u optimalizační úlohy nejdříve zamyslet, zda umíte vyřešit rozhodovací variantu s pevným parametrem), platíme za něj logaritmem. Někdy ovšem existuje i řešení, které ten logaritmus ušetří.

Pokud spustíme hladový algoritmus s parametrem $k = n$, tak víme, že v čase $\mathcal{O}(n \log n)$ vždy vrátí nějaké rozdělení na n podposloupností, přičemž některé z nich mohou být prázdné. Nechť k' je počet neprázdných podposloupností v tomto rozdělení. Všimněte si, že jediné místo, kde ve hladovém algoritmu hraje roli parametr, je ve chvíli, kdy chceme umístit prvek do nové (prázdné) posloupnosti a zjišťujeme, zda ještě nějakou takovou máme k dispozici. To znamená, že hladový algoritmus vrátí to samé rozdělení pro libovolnou hodnotu parametru větší nebo rovnu k'.

Zároveň pro hodnoty parametru menší než k' hladový algoritmus odpoví, že řešení neexistuje, a výše jsme dokázali, že v takovou chvíli opravdu neexistuje. To znamená, že k' je nejmenší počet rostoucích podposloupností, na něž zadaná posloupnost lze rozdělit. Tím jsme úlohu vyřešili v čase $\mathcal{O}(n \log n)$ a získali tak plný počet bodů. Paměťová složitost je $\mathcal{O}(n)$.

Poznámka

Při lepší implementaci hladového algoritmu mu vůbec nemusíme zadávat parametr k a místo toho ho nechat vždy, když potřebuje, vytvořit novou prázdnou posloupnost a vložit do ní aktuální prvek. Díky tomu potom binárně vyhledáváme pouze v již vytvořených posloupnostech, kterých je vždy nejvýše k', a tedy složitost zlepšíme na $\mathcal{O}(n \log k')$. Toto však nebylo pro zisk plného počtu bodů nutné.

Dilworthova věta

Tato úloha blízce souvisí s takzvanou Dilworthovou větou o částečných uspořádáních. *Částečné uspořádání* je zobecnění lineárního uspořádání, kde nevyžadujeme, aby každá dvojice prvků byla porovnatelná (ale stále platí, že pokud $a \leq b$ a $b \leq a$, potom $a = b$, a také tranzitivita, tj. je-li $a \leq b$ a $b \leq c$, pak i $a \leq c$). Pokud je $(P, \leq)$ částečné uspořádání, a $X \subseteq P$, potom je X *řetězec*, pokud každé dva prvky X jsou navzájem porovnatelné, a X je *antiřetězec*, pokud žádné dva prvky X nejsou porovnatelné. Dilworthova věta říká, že velikost největšího antiřetězce v $(P, \leq)$ je rovna nejmenšímu počtu řetězců, na něž lze $(P, \leq)$ rozdělit.

Máme-li posloupnost různých čísel $a_1, \ldots, a_n$ ze zadání, můžeme na množině $P = \{a_1, \ldots, a_n\}$ definovat částečné uspořádání $\preceq$ tak, že $a_i \preceq a_j$ právě tehdy, když $a_i \leq a_j$ a zároveň $i \leq j$, tj. porovnatelné zůstanou právě rostoucí dvojice. Úloha po nás potom chce přesně rozklad tohoto nového uspořádání na řetězce. Dilworthova věta říká, že nejmenší možný počet takových řetězců bude rovna velikosti největšího antiřetězce. Ovšem antiřetězce přesně odpovídají klesajícím podposloupnostem. Řečeno v původním jazyce úlohy, nejmenší počet rostoucích podposloupností, na něž lze zadaná posloupnost rozdělit, je roven délce nejdelší klesající podposloupnosti. Pokud znáte algoritmus na její hledání (který je z určitého pohledu podobný našemu hladovému algoritmu) a Dilworthovu větu, mohli jste získat 8 bodů.

```
#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];

        // Podposloupnosti budou vždy seřazené sestupně podle posledního prvku
        vector<vector<int>> podposloupnosti;

        // Umístíme první prvek.
        podposloupnosti.push_back({a[0]});

        for (int i = 1; i < n; ++i) {
                // Binárně vyhledáme vhodnou posloupnost
                int l = 0, r = podposloupnosti.size();
                while (r - l > 1) {
                        int m = (r + l)/2;
                        if (podposloupnosti[m].back() < a[i]) r = m;
                        else l = m;
                }
                // Na konci je v l index poslední posloupnosti,
                // jejíž poslední prvek je větší než a[i],
                // anebo l = 0 a a[i] je větší než všechny prvky,
                // které jsme doteď viděli
                if (l == 0 && podposloupnosti[l].back() < a[i]) --l;

                if (l + 1 == podposloupnosti.size()) {
                        // Musíme založit novou posloupnost
                        podposloupnosti.push_back({a[i]});
                } else {
                        podposloupnosti[l+1].push_back(a[i]);
                }
        }

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

        return 0;
}
```

P-II-2 Developerský projekt

Na vstupu máme strom T, jehož vrcholy jsou stanice a hrany odpovídají sousednosti na některé z linek. Úloha se nás ptá na počet dvojic vrcholů, které jsou od sebe vzdálené $k+1$ (nejkratší cesta mezi nimi obsahuje $k+1$ hran).

Ty můžeme nalézt standardními algoritmy na hledáním nejkratších cest v grafech – mezi každými vrcholy vede právě jedna cesta, a délku této cesty nám tyto algoritmy naleznou. Vzdálenosti všech dvojic vrcholů můžeme nalézt buď Floydovým–Warshallovým algoritmem (viz kuchařka KSP o dynamickém programování *http://ksp.mff.cuni.cz/viz/kucharky/dynamika*) v čase $\mathcal{O}(n^3)$, nebo spuštěním prohledávání do šířky z každého vrcholu s celkovou složitostí $\mathcal{O}(n^2)$. Jakmile máme vzdálenosti všech dvojic vrcholů, stačí jen spočítat počet dvojic, které mají vzdálenost přesně $k+1$. Celkově tedy bude naše řešení kubické nebo kvadratické v závislosti na použitém algoritmu a získá 3 nebo 5 bodů.

Rychlejší řešení

Pokud ale chceme získat více bodů, nemůžeme si dovolit počítat vzdálenosti mezi všemi dvojicemi – těch je kvadraticky mnoho. Místo toho použijeme techniku, které se říká *dynamické programování na stromech.*

Zakořeňme si strom v libovolném vrcholu. V každé cestě v T potom můžeme najít její jednoznačně určený *nejvyšší vrchol* (tj. vrchol nejblíže ke kořeni). V našem algoritmu pak každou cestu délky $k+1$ započítáme právě v jejím nejvyšším vrcholu.

Představme si nyní, že jsme ve vrcholu v, označme jeho potomky $w_1, \ldots, w_n$ a podstromy v nich zakořeněné jako $T_1, \ldots, T_n$. Každá cesta, jejímž je v nejvyšším vrcholem, začíná v nějakém vrcholu x v nějakém podstromu T_i, potom vyšplhá do vrcholu v a tam buď skončí, anebo sestoupí do vrcholu y v nějakém jiném podstromu T_j pro $j \neq i$. Délka této cesty je potom součet vzdálenosti x od w_i, vzdálenosti w_i od v (což je 1), a pokud cesta ve v nekončí, tak ještě musíme přičíst 1 za hranu z v do w_j a potom vzdálenost y od w_j.

Pro libovolný vrchol v a číslo i označme jako $p(v, i)$ počet vrcholů v podstromu zakořeněném ve v, které mají od v vzdálenost i.

Cesty, které mají délku $k+1$ a mají nejvyšší vrchol v (každá taková cesta odpovídá požadované dvojici vrcholů) můžeme rozdělit do dvou typů: ty, které mají v jako koncový vrchol, a ty, které nikoliv. První typ započteme snadno: od každého syna víme, kolik vrcholů v daném podstromu je ve vzdálenosti $k+1$ od v. Ty stačí jen sečíst, což zvládneme v čase lineárním s počtem synů.

Trochu složitější to bude s cestami druhého typu. To jsou všechny cesty mezi nějakými dvěma vrcholy a a b takovými, že jsou v jiném podstromu a navíc je-li vzdálenost z a do v nějaké i, pak vzdálenost z b do v musí být $k+1-i$.

To můžeme přímočaře udělat tak, že vyzkoušíme všechny dvojice podstromů a sečteme pro každé i od 0 do $k+1$ počet vrcholů z prvního podstromu, které mají vzdálenost do v i, krát počet vrcholů z druhého, které mají vzdálenost $k+1-i$. Tím v každém vrcholu strávíme $\mathcal{O}(d_v^2 k)$, kde d_v je počet sousedů sousedů daného vrcholu.

Zbývá nám spočítat hodnoty $p(v, i)$. Zřejmě nám to stačí pro $0 \leq i \leq k$, cesty délky $k+1$ můžeme rovnou započítat a nemusíme si je dál pamatovat. Je-li v list, pak $p(v, 0) = 1$ a $p(v, i) = 0$ pro všechna $i > 0$. Jinak v není list. Předpokládejme, že již máme spočítané $p(w, i)$ pro každého potomka w vrcholu v. Potom sečteme hodnoty, které jsme dostali ze synů v, a přidáme novou cestu o délce 0.

V jednom vrcholu v tedy bude výpočet probíhat následovně:

```
odpoved = 0

for (x : potomek(v))
    odpoved += p(x, k)

for (x : potomek(v)) for(y : potomek(v))
    if (x < y) // Každou dvojici chceme započítat jen jednou
        for (int i = 0; i <= k - 1; ++i)
            odpoved += p(x, i) * p(y, k - 1 - i)
```

```
p(v, i) = 0
for (x : potomek(v))
    for (int i = 0; i < k; ++i)
        p(v, i + 1) += p(x, i)
p(v, 0) = 1
```

Celý tento výpočet implementujeme pomocí jednoho prohledání do hloubky z kořene. Vždy nejprve navštívíme a zpracujeme všechny potomky aktuálního vrcholu, potom s pomocí jejich hodnot p spočítáme počet cest délky $k+1$, jejichž je aktuální vrchol nejvyšším vrcholem, a nakonec tyto hodnoty využijeme i k výpočtu hodnot p pro aktuální vrchol.

V každém vrcholu strávíme čas $\mathcal{O}(d_v^2 k)$, tedy celkem $\mathcal{O}(d_{\max}^2 nk)$. Takové řešení stačilo pro zisk 8 bodů. (Ve skutečnosti lze odhad udělat lépe a dostat $\mathcal{O}(d_{\max} nk)$, ale z hlediska zisku bodů to nepomůže.)

Velké stupně

Pokud budou stupně velké, předchozí řešení není o nic rychlejší než opakované prohledávání do šířky. Zaměřme se proto na část, která trvá nejdéle – počítání cest, které jdou mezi různými podstromy. Ideálně bychom chtěli tento počet získat jedním průchodem, nikoliv procházením všech dvojic podstromů.

Všimněme si toho, že když sčítáme cesty daných délek pro jednotlivé podstromy při výpočtu $p(v,i)$, tak před tím než započteme podstrom j-tého syna, známe pro každé i součet počtů cest délky i nacházejících se v podstromech všech synů až do $(j-1)$-ho. Můžeme tedy snadno spočítat počet cest délky $k+1$ mezi vrcholy z j-tého podstromu a vrcholy z podstromy před ním. Když toto sečteme pro všechna j, započteme tak každou cestu právě jednou, čímž složitost tohoto kroku snížíme na $\mathcal{O}(d_v k)$. (Zbavit se kvadratické složitosti na počtu synů se dalo i jinými způsoby. Například, že spočteme počet všech cest v podstromu určeném vrcholem v, tedy i těch zdegenerovaných, a následně od tohoto počtu odečteme počet cest, které jsou uvnitř nějakého z podstromů.)

```
for (int i = 0; i <= k; ++i)
    p(v, i) = 0

p(v, 0) = 1
for (x : potomek(v))
    for (int i = 0; i <= k; ++i)
        odpoved += p(v, i) * p(x, k - i)
    for (int i = 0; i < k; ++i)
        p(v, i + 1) += p(x, i)
```

Ve vrcholu v uděláme $\mathcal{O}(d_v k)$ práce, tedy pokud opět použijeme prohledávání do hloubky, uděláme celkem $\mathcal{O}(k \sum_v d_v) = \mathcal{O}(nk)$ práce. Zde využíváme, že součet stupňů je dvojnásobek počtu hran, což je v našem případě $2n-2$. Tedy časová složitost je $\mathcal{O}(nk)$, stejně tak paměťová, což již stačí na plný počet bodů.

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

vector<vector<int>> al;
int k_plus_1;
long long answer = 0;

vector<int> dfs (int index, int parent) {
        vector<int> ret(k_plus_1 + 1);
        for (int son : al[index]) if (son != parent) {
                auto got = dfs(son, index);
                answer += got[k_plus_1];
                for (int i = 0; i < k_plus_1 + 1; ++i) {
                        answer += got[i] * ret[k_plus_1 - i];
                        ret[i] += got[i];
                }
        }

        // Posuneme hodnoty o jednu pozici.
        for(int i = k_plus_1 - 1; i >= 0; --i) {
                ret[i + 1] = ret[i];
        }

        // Přidáme aktuální stanici.
        ret[1]++;
        return ret;
}

int main() {
        ios_base::sync_with_stdio(0);

        // Přečteme vstup.
        int n, k; cin >> n >> k;
        k_plus_1 = k + 1;
        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;
        return 0;
}
```

Jiné řešení

Úloha šla také vyřešit v čase $\mathcal{O}(n \log n)$ (bez závislosti na k) pomocí takzvané *centroidové dekompozice* (viz například vzorové řešení úlohy 33-2-X1 z KSP, které najdete na *https://ksp.mff.cuni.cz/viz/33-2-X1/reseni*). *Centroid* stromu je takový vrchol, že po jeho odebrání se strom rozpadne na komponenty, každá z nichž má nejvýše $n/2$ vrcholů. Máme-li nějaký centroid, můžeme v něm strom zakořenit, v každém podstromu daném jeho potomkem pomocí prohledávání do šířky či hloubky spočítat počty vrcholů ve všech vzdálenostech od centroidu a potom stejným způsobem jako v našem řešení pomocí těchto počtů v čase $\mathcal{O}(n)$ spočítat počet

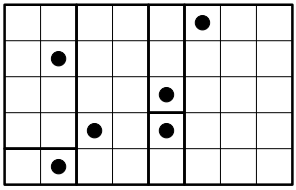
cest délky $k+1$, které procházejí vybraným centroidem. Počty cest, které celé leží v jednotlivých podstromech, můžeme spočítat rekurzivním voláním. Tím v každé hladině rekurze uděláme $\mathcal{O}(n)$ práce, a protože se v každé hladině velikost problému zmenší minimálně na polovinu, po $\mathcal{O}(\log n)$ rekurzivních voláních dostaneme nejvýše jednoprvkové stromy, v nichž úlohu umíme vyřešit triviálně.

P-II-3 Brusinková čokoláda

Nejprve se pojďme zamyslet nad variantou, kdy nemusíme optimalizovat velikost dílku pro Dana. Tedy stačí rozdělit čokoládu na obdélníky tak, aby v každém byl přesně jeden brusinkový dílek.

Začneme tím, že čokoládu rozlámeme na části tak, aby v každé části byl jen jeden sloupec obsahující nějaké brusinky. To můžeme například zařídit tak, že čokoládu rozlomíme před každým sloupcem s brusinkami až na první.

Každou z těchto částí již budeme řešit samostatně. Podobným algoritmem ji rozdělíme na obdélníky, kde každý bude obsahovat jeden řádek s brusinkami. Ovšem jelikož také každý obdélník bude obsahovat pouze jeden sloupec s brusinkami, musí v něm být právě jeden brusinkový dílek. I toto zařídíme jednoduše. Zase nám stačí rozlomit čokoládu před každým řádkem s brusinkou (v dané části) až na první.

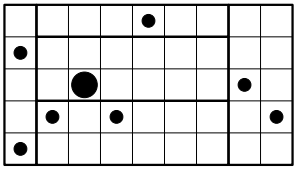


Již tedy víme, že řešení vždy existuje, a nikdy tedy nebude potřeba vypsat `NELZE`. Ještě pojďme dorozmyslet implementační detaily, tedy jak vypsat výsledné obdélníky. Nejprve si setřídíme brusinky podle sloupce a rozdělíme je do skupin. Podle okolních skupin již víme, kde bude levý a pravý okraj obdélníků brusinek ve skupině. Každou skupinu pak setřídíme podle řádku a pro každou brusinku jíž snadno dopočteme i horní a dolní okraj podle okolních brusinek.

Časová složitost takového řešení je $\mathcal{O}(n \log n)$. Pomocí přihrádkového třídění ji lze změnit na $\mathcal{O}(rs)$. Paměťová složitost je $\mathcal{O}(n)$ resp. $O(n+r+s)$ v případě přihrádkového třídění.

Nyní se pojďme zamyslet nad plnou verzí úlohy: Na chvíli si představme, že již si Dan vybral svůj největší obdélník. Zvládneme rozdělit zbytek? Můžeme oddělit všechny sloupce nalevo a napravo od Danova dílku. Ze zbytku pak oddělíme ještě všechny řádky nad a pod Danovým obdélníkem. Může se stát, že v některém směru bude Danův díl až po okraj. Pak nic neodlomíme. Pokud ale něco odlomíme, tak víme, že v odlomené části bude brusinkový dílek. Kdyby tam totiž žádný nebyl, tak by si Dan svůj dílek mohl rozšířit až po okraj a tím by získal víc čokolády. To se však stát nemůže, protože mi předpokládáme, že si Dan vybral největší dílek.

Každou z odlomených částí tedy můžeme vyřešit jako samostatnou tabulku pomocí předešlého algoritmu. Víme, že pokud obsahuje alespoň jeden brusinkový dílek, tak vždy zvládneme najít řešeni.



Zbývá tedy poslední věc: najít Danovi nejlepší kousek čokolády. Pro zisk alespoň nějakých bodů stačilo vyzkoušet všech $\mathcal{O}(r^2s^2)$ možností Danova dílku a pro každý dílek zjistit, zdali obsahuje pouze největší brusinku. Z takových pak vybrat ten největší. Celková časová složitost řešení je $\mathcal{O}(r^2s^2n)$, protože pro každý čtverec zjišťujeme, co je uvnitř, v čase $\mathcal{O}(n)$.

Pojďme zrychlit zjišťování, zdali je v obdélníku i jiný brusinkový dílek. Můžeme využít dvojrozměrných prefixových součtů (viz kuchařka KSP Základní algoritmy *https://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy/*). Prostě spočítáme počet brusinkových dílků uvnitř a když je to 1, tak si daný obdélník může Dan nechat.

Pojďme se podívat jaký je největší možný obdélník pro nějaký zafixovaný levý a pravý okraj. Pokud na řádku s největší brusinkou je mezi ní a okraji ještě jiná brusinka, tak žádný takový obdélník neexistuje. V opačném případně můžeme zkusit obdélník zvyšovat nahoru, dokud nenarazíme na nějakou brusinku. Stejným způsobem pak můžeme rozšířit směrem dolů. V každém sloupci si můžeme tedy spočítat, jaká je vzdálenost do nejbližší překážky (brusinky nebo okraje) směrem nahoru a dolu. Ze všech sloupců mezi okraji pak stačí v obou směrech vzít minimum a tím máme určený největší obdélník pro daný levý a pravý okraj.

Pro každý sloupec si tedy můžeme nejprve přepočítat volný prostor v obou směrech. Pak jen vyzkoušíme všechny možné dvojice levých a pravých okrajů. Ovšem místo počítáním minim pro každý zvlášť na to půjdeme chytřeji. Všimneme si, že když obdélník rozšíříme o jeden sloupec doprava, tak stačí dosavadní minima v obou směrech jednoduše aktualizovat. Pro každý levý okraj tedy snadno a rychle spočteme všechny pravé okraje. Celková časová složitost tedy je $\mathcal{O}(rs + s^2)$, což už stačí na plný počet bodů. Paměťová složitost je při vhodné implementaci $\mathcal{O}(r + s + n)$.

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

using namespace std;

void obdelnik(int x_min, int y_min, int x_max, int y_max)
{
        // Vytiskne požadovaný obdélník
        cout << x_min << " " << y_min << " " << x_max << " " << y_max << "\n";
}
```

```
void vyres_sloupec(vector<pair<int, int>> brusinky,
                   int x_min, int  y_min, int x_max, int y_max)
{
        // Funkce předpokládá, že všechny brusinky na vstupu leží v jednom sloupci
        sort(brusinky.begin(), brusinky.end());
                        // Setřídíme brusinky podle x-ové souřadnice
        brusinky.push_back({x_max+1, 0});
                        // Na konec ještě přidáme jednu virtuální brusinku,
                        // která nám správně vyřeší konec posledního obdélníku
        obdelnik(x_min, y_min, brusinky[1].first-1, y_max);
        for (int i = 1; i < brusinky.size()-1; i++)
                obdelnik(brusinky[i].first, y_min, brusinky[i+1].first-1, y_max);
}

void vyres_bez_dana(vector<pair<int, int>> brusinky,
                    int x_min, int y_min, int  x_max, int y_max)
{
        if (x_min > x_max || y_min > y_max) return;
                        // Když je obdélník plochý, není co dělat
        vector<vector<pair<int, int>>> brusinky_ve_sloupci(y_max - y_min + 1);
        for (auto it: brusinky)
                if (x_min <= it.first && it.first <= x_max &&
                    y_min <= it.second && it.second <= y_max)
                        // (odfiltrovali jsme brusinky mimo daný obdélníček)
                        brusinky_ve_sloupci[it.second-y_min].push_back(it);
        int zacatek_dilku = y_min;
        int y = y_min;
        while (brusinky_ve_sloupci[y-y_min].size() == 0) y++;
        while (y <= y_max)
        {
                int dalsi_sloupec = y+1; // y_max+1 pokud další již není
                while (dalsi_sloupec <= y_max
                                && !brusinky_ve_sloupci[dalsi_sloupec-y_min].size())
                        dalsi_sloupec++;
                vyres_sloupec(brusinky_ve_sloupci[y-y_min],
                              x_min, zacatek_dilku, x_max, dalsi_sloupec-1);
                zacatek_dilku = dalsi_sloupec;
                y = dalsi_sloupec;
        }
}

void vyres(vector<pair<int, int>> brusinky,
           int x_min, int y_min, int  x_max, int y_max)
{
        int dan_x = brusinky[0].first;
        int dan_y = brusinky[0].second;

        vector<int> prostor_nahoru(y_max - y_min + 1, dan_x - x_min + 1);
        vector<int> prostor_dolu(y_max - y_min + 1, x_max - dan_x + 1);
                // 1 značí, když může byt obdélník alespoň na řádku s brusinkou
                // 0 značí, že tady již obdélník být nemůže

        for (auto it: brusinky)
        {
                if (it == brusinky[0])
                        continue;
                int index = it.second - y_min;
```

```
                if (it.first <= dan_x)
                        prostor_nahoru[index] =
                                min(prostor_nahoru[index], dan_x - it.first);
                if (it.first >= dan_x)
                        prostor_dolu[index]   =
                                min(prostor_dolu[index], it.first - dan_x);
        }
        int opt_velikost = -1;
        int opt_i, opt_j, opt_nahoru, opt_dolu;
        for (int i=y_min; i<=dan_y; i++)
        {
                int aktualne_nahoru = 1<<30;
                int aktualne_dolu = 1<<30; // Nastavíme na nekonečno
                for (int j=i; j<=y_max; j++)
                {
                        aktualne_nahoru =
                                min(aktualne_nahoru, prostor_nahoru[j-y_min]);
                        aktualne_dolu   =
                                min(aktualne_dolu  , prostor_dolu[j-y_min]  );
                        int velikost = (aktualne_nahoru + aktualne_dolu - 1)
                                * (j - i + 1);
                        if (j >= dan_y && velikost > opt_velikost)
                        {
                                opt_velikost = velikost;
                                opt_i = i, opt_j = j;
                                opt_nahoru = aktualne_nahoru;
                                opt_dolu = aktualne_dolu;
                        }
                }
        }
        obdelnik(dan_x-opt_nahoru+1, opt_i, dan_x+opt_dolu-1, opt_j);
        vyres_bez_dana(brusinky, x_min, y_min, x_max, opt_i-1);
        vyres_bez_dana(brusinky, x_min, opt_j+1, x_max, y_max);
        vyres_bez_dana(brusinky, x_min, opt_i, dan_x - opt_nahoru, opt_j);
        vyres_bez_dana(brusinky, dan_x + opt_dolu, opt_i, x_max, opt_j);
}

int main() {
        int r, s, n;
        vector<pair<int, int>> brusinky;

        cin >> r >> s >> n;
        brusinky.resize(n);
        for (int i = 0; i < n; ++i)
                cin >> brusinky[i].first >> brusinky[i].second;

        vyres(brusinky, 1, 1, r, s);
        return 0;
}
```

P-II-4 Hvězdní věštci

Část a)

Hlavní ideou řešení je, že každý systém si nechá vyvěštit, v kolika systémech, které jsou před ním na cestě, probíhá povstání. Toto číslo pak zvýší o jedna, pokud v něm povstání probíhá, a pošle ho systému, který po něm následuje. Každý systém ověří, zda je jeho věštba konzistentní, tj. zda od svého předchůdce obdržel stejné číslo jako to, které vyvěštil. Poslední systém na cestě pak ověří, zda celkový počet povstání je alespoň $N/3$. Drobným technickým detailem je, že věštec nám také musí vyvěštit, který z našich sousedů je na cestě před námi, a musíme obdobě zajistit, že tato volba je konzistentní s tou v sousedních systémech. Na to stačí jeden bit, celkem tedy máme $K = \lceil \log_2 N \rceil + 1$.

V pseudokódu vypadá řešení následovně:

```
struct {
    unsigned(ceil(log(N))) pocet_povstani;
    unsigned(1) naslednik;
  } R;

struct {
    unsigned pocet_povstani;
    bool od_naslednika;
  } send[D], receive[D];

/* Speciální případ. */
if (N == 1)
  return P ? ANO : NE;

/* První vrchol na cestě. */
if (D == 1 && R.naslednik == 0)
  {
    send[0].pocet_povstani = P ? 1 : 0;
    send[0].od_naslednika = false;

    return receive[0].od_naslednika ? ANO : NE;
  }

unsigned povstani_vcetne = R.pocet_povstani + (P ? 1 : 0);

/* Odešleme informace předchůdci, a pokud nejsme poslední vrchol,
   i následníkovi. */
send[1 - R.naslednik].od_naslednika = true;
if (D == 2)
  {
    send[R.naslednik].od_naslednika = false;
    send[R.naslednik].pocet_povstani = povstani_vcetne;
  }

/* Systém, kterého jsme uhodli jako předchůdce, by si neměl myslet,
   že je následník. */
if (receive[1 - R.naslednik].od_naslednika)
  return NE;

/* My a náš předchůdce bychom měli mít stejný názor na počet
   povstání před námi. */
if (receive[1 - R.naslednik].pocet_povstani != R.pocet_povstani)
  return NE;
```

```
if (D == 2)
  {
    /* Systém, kterého jsme uhodli jako následníka, by si neměl myslet,
       že je předchůdce. */

    return receive[R.naslednik].od_naslednika ? ANO : NE;
  }

/* Ověření počtu v posledním systému na cestě. */
return pocet_vcetne >= N/3 ? ANO : NE;
```

Jestliže povstání probíhá v alespoň $N/3$ systémech, věštci jistě mohou přežít tak, že všechny informace (kdo je následník, kolik povstání probíhá před námi) vyvěští správně.

Naopak, pokud všechny systémy odpoví ANO, znamená to, že za prvé se každé dva sousední systémy shodly na tom, který z nich následuje na cestě dřív, a speciálně jeden ze systémů s jediným sousedem byl určen jako začátek cesty a druhý jako konec. A za druhé, že počet povstání, který každý systém vyvěštil, je roven tomu, který vyvěštil jeho předchůdce, navýšený o jedna, pokud v předchůdci probíhá povstání. Z toho plyne, že každý systém vyvěštil přesně počet povstání, které probíhají na cestě před ním. Jelikož poslední systém odpověděl ANO, znamená to tedy, že počet povstání je alespoň $N/3$.

Část b)

Jedním z možných řešení je, že každý hvězdný systém vyvěští číslo R mezi 0 a $N-1$ (je tedy potřeba $K = \lceil \log_2 N \rceil$), pošle ho všem svým sousedům a odpoví ANO, právě když od nejvýše jednoho ze sousedů obdrží číslo menší nebo rovné R. V pseudokódu:

```
unsigned(ceil(log(N))) R, send[D], receive[D];
for (i = 0, ..., D - 1)
  send[i] = R;

int pocet_mensich_nebo_rovnych = 0;
for (i = 0, ..., D - 1)
  if (receive[i] <= R)
    pocet_mensich_nebo_rovnych++;

return pocet_mensich_nebo_rovnych <= 1 ? ANO : NE;
```

Proč toto řešení funguje? Pokud je Hvězdné impérium strom, povšimněme si, že z každého systému vede jednoznačná cesta (posloupnost sousedících systémů) do systému číslo 0, a věštec může jako R vyvěštit délku této cesty. Systém číslo 0 pak obdrží 1 ode všech sousedů, zatímco každý jiný systém obdrží číslo $R-1$ od svého souseda na cestě do systému 0 a $R+1$ od všech svých dalších sousedů. Všechny systémy tedy odpoví ANO.

Naopak, pokud Hvězdné impérium obsahuje kružnici C, podívejme se na systém na C, kterému věštec vyvěštil ze všech systémů v C největší hodnotu R (pokud je takových systémů víc, pak na libovolný z nich). Tento systém obdrží od svých dvou sousedů na C čísla menší nebo rovná R, a odpoví tedy NE.