P-III-1 Dobíjecí stanice

Řešení budeme popisovat v terminologii teorie grafů: vrcholy odpovídají městům, silnice hranám a víme, že graf je neorientovaný, nehodnocený a navíc je to strom. V některých vrcholech jsou navíc dobíjecí stanice.

Nejpřímočařejší myšlenka je prostě zvlášť vyzkoušet každý vrchol jako počáteční, prozkoumat, do kolika vrcholů jsme z něj schopni dojet, a tyto počty pak posčítat přes všechny počáteční vrcholy. Mějme tedy nějaký počáteční vrchol v_0. Jak najít z něj dosažitelné vrcholy? Obyčejné prohledávání do hloubky nebo do šířky nestačí, protože se může stát, že nějaký vrchol potřebujeme navštívit vícekrát, viz například druhý příklad ze zadání: cesta 1–2–3–5–6 s dobíjecí stanicí ve vrcholu 4, který je napojen na vrchol 3. Pro $k = 3$ a počátek ve vrcholu 1 jsme schopni dojet do vrcholu 6 pouze s mezizastávkou ve vrcholu 4, čímž vrchol 3 projedeme dvakrát.

Jedna možnost, jak z toho ven, je použít podobný trik s nafukováním grafu jako v úlohách Vánoční návštěvy z krajského kola a Přívalový déšť z domácího kola. Vyrobíme nový orientovaný graf čítající celkem $n \cdot (k + 1)$ vrcholů (pro odlišení od původních vrcholů jim budeme říkat *stavy*), přičemž stav (v, b) pro $v \in \{1, \ldots, n\}, b \in \{0, \ldots, k\}$ reprezentuje situaci, kdy stojíme ve vrcholu v a v baterii zbývá energie na b silnic. Hrany vytvoříme následovně: ze stavů tvaru $(v, 0)$ žádná hrana nepovede a pro stavy (v, b) s $b \geq 1$ se podíváme na všechny hrany vw, které z v vedou v původním grafu, a za každou z nich vytvoříme orientovanou hranu z (v, b) do $(w, b - 1)$ (pokud ve w nestojí dobíjecí stanice), nebo z (v, b) do (w, k) (pokud tam dobíjecí stanice stojí). Rozmyslíme si, že takový graf přesně modeluje Matějovo možné chování v původním grafu, takže nyní stačí spustit prohledávání ze stavu (v_0, k) a spočítat, do kolika různých vrcholů původního grafu se umíme dostat.

Nafouknutý graf má $\mathcal{O}(nk)$ stavů i hran a tolik času nás stojí i jeho vybudování a jedno projití. Můžeme si ho vybudovat na začátku programu, to však nic nemění na tom, že ho musíme n-krát projít. Toto řešení tedy běží v čase $\mathcal{O}(n^2k)$ a šly za něj získat až 4 body.

Zrychlujeme

Abychom dosáhli lepší časové složitosti, musíme se nafouknutého grafu zase zbavit. U myšlenky vyzkoušení všech počátečních vrcholů však ještě zůstaneme. Naivní prohledávání v původním grafu se sice rozbije kvůli nutnosti navštěvovat vrcholy vícekrát, ale ono navštěvování nemusí být zase tak nahodilé. Konkrétně si za chvíli rozmyslíme, že stačí pouštět upravené prohledávání do hloubky v původním grafu, které každý vrchol navštíví nejvýše jednou a během toho si u každého vrcholu v pamatuje aktuální stav baterie $b(v)$ (nebo -1, pokud se do v neumíme dostat), s následující úpravou: po vstupu do vrcholu si před dalším prohledáváním zaskočíme na nejbližší dobíjecí stanici (potenciálně ještě v neobjevené části grafu) a pak zase

zpátky, ledaže by nám na to nestačila baterie nebo bychom si oproti původnímu nabití pohoršili. Důležité je, že tyto zajížďky nepočítáme do prohledávání, ani při nich neznačíme vrcholy jako navštívené.

Nejprve si povězme, jak nejbližší stanici hledat. Vzhledem k tomu, že nás nezajímá konkrétní trasa, ale pouze zda se do nějakého vrcholu umíme dostat, stačí nám pro každý vrchol v spočítat jen jeho vzdálenost $d(v)$ od nejbližší dobíjecí stanice. To provedeme pro všechny vrcholy naráz pomocí upraveného prohledávání do šířky puštěného zároveň ze všech dobíjecích stanic: nejprve si všechny dobíjecí stanice uložíme do fronty a položíme jim $d(v) = 0$ a poté postupně vybíráme vrcholy z fronty a všem ještě nenavštíveným sousedům nastavujeme $d(s) = d(v) + 1$ a přidáváme je na konec fronty.

S pomocí $d(v)$ můžeme „zaskočení si“ na nejbližší dobíjecí stanici popsat takto: pokud $b(v) \leq d(v)$ (nemáme dost energie) nebo $k - d(v) \leq b(v)$ (dobitím bychom si nepomohli), neděláme nic, jinak nastavíme $b(v) \leftarrow k - d(v)$ (dobijeme si a pak jedeme $d(v)$ silnic zpátky).

A proč tedy algoritmus funguje? Vše bude jasné, když si rozmyslíme, že $b(v)$ nepočítá jen tak nějaké nabití baterie při nějakém konkrétním průjezdu, počítá totiž *nejlepší možné* nabití, se kterým se do vrcholu v umíme dostat. To si můžeme rozmyslet sporem: nechť pro spor existuje nějaký vrchol v a sled z v_0 do v (který potenciálně navštíví v víckrát) takový, že po jeho projití skončíme ve vrcholu v s nabitím $B > b(v)$. Ze všech takových v si vybereme ten, na který prohledávání narazí nejdříve. Pro v_0 zjevně problém nenastane, jelikož $b(v_0) = k$ a lepší nabití z principu neumíme. Nechť jsme tedy přišli do nějakého vrcholu $v \neq v_0$. Klíčové je pozorování, že náš graf je strom, takže libovolný sled do v povede přes nějakého jeho souseda s, který již má $b(s)$ správně spočítané. Zároveň platí $b(v) \geq b(s) - 1$, protože prohledávání nastaví tuto hodnotu při příchodu do v a pak ji možná zlepší.

Podívejme se na nyní okamžik, kdy náš sled poprvé vstupuje do vrcholu v. V tom okamžiku muselo být nabití nejvýše $b(s) - 1$, protože do v vstupujeme z s, v němž jsme nemohli mít větší nabití než $b(s)$. Jelikož platí $b(v) \geq b(s) - 1$, musí sled po prvním vstupu do v navštívit alespoň jednu dobíjecí stanici, jinak by platilo $B \leq b(s) - 1 \leq b(v)$. Podívejme se na konec sledu, konkrétně na úsek od poslední dobíjecí stanice do konce. V dobíjecí stanici bylo nabití rovno k a pak následovalo alespoň $d(v)$ kroků, abychom se z dobíjecí stanice dostali zpátky do v; tedy platí $B \leq k - d(v)$. To je ale spor, protože v takovém případě jsme během prohledávání rovněž zjistili, že zároveň umíme a zároveň se nám vyplatí zajet do nejbližší dobíjecí stanice a nastavit $b(v) = k - d(v)$.

Tím je správnost algoritmu dokázána, jelikož podle předchozího důkazu platí $b(v) \geq 0$ právě tehdy, když do vrcholu v vede trasa. Jaká je časová složitost? Spočítání pole d můžeme provést v $\mathcal{O}(n)$ na začátku algoritmu. Poté n-krát pouštíme prohledávání do šířky, časovou složitost jsme tedy vylepšili na $\mathcal{O}(n^2)$. Za takovéto řešení se dalo získat až 6 bodů.

Počítáme vše najednou

V optimálním řešení se oprostíme od pouštění prohledávání z každého vrcholu zvlášť a místo toho se pokusíme všechny vyhovující dvojice spočítat najednou. K tomu si strom zakořeníme za libovolný vrchol. Zopakujeme si názvosloví: nyní má každý vrchol několik (možná i nula) *synů* a právě jednoho *otce* (až na kořen, ten otce nemá). Vrcholy tvaru „otec v, otec otce v, ...“ se souhrnně nazývají *předkové/předchůdci* vrcholu v, vrcholy tvaru „syn v, syn syna v, ...“, jsou *potomci* vrcholu v. Vrchol v spolu se všemi svými potomky tvoří *podstrom* pod vrcholem v.

Nejprve si rozmyslíme pomocné tvrzení, které se nám bude hodit později: pro vrcholy a a b platí, že můžeme vyrazit z vrcholu a s nabitím A a dorazit do vrcholu b s nabitím aspoň B, právě když jde vyrazit z vrcholu b s nabitím $k - B$ a dorazit do vrcholu a s nabitím aspoň $k - A$. Vlastně nejde o nic těžkého, stačí si uvědomit, že můžeme vzít sled z a do b a obrátit ho, a rozmyslet si že původní sled splňuje podmínky s A a B právě tehdy, když obrácený sled splňuje podmínky s $k-B$ a $k-A$. K tomu si stačí sled rozložit na úseky podle návštěv dobíjecích stanic a uvědomit si, jak jsou které úseky dlouhé.

S tímto tvrzením zadarmo dostáváme, že pokud se umíme s nabitou baterií dopravit z a do b, pak se také umíme s nabitou baterií dopravit z b do a, tj. všech vyhovujících dvojic měst bude vždy sudý počet.

Zaveďme ještě jedno značení: nechť $b(u \to v)$ označuje nejvyšší možné nabití baterie, které můžeme dostat ve vrcholu v, pokud vyrazíme z vrcholu u s plným nabitím a dorazíme do vrcholu v (přičemž vrchol v můžeme navštívit několikrát, než v něm zastavíme). Jinými slovy, pokud bychom z vrcholu u spustili prohledávání z kvadratického řešení, označuje $b(u \to v)$ právě hodnotu $b(v)$.

Počet vyhovujících dvojic spočítáme dynamickým programováním. Strom budeme procházet do hloubky a pokaždé, když se z nějakého vrcholu v budeme vracet, využijeme informace spočítané v jeho synech ke spočítání informací v něm. Konkrétně pro každé v budeme počítat pole $C_v[0 \dots k]$, kde $C_v[b]$ bude označovat počet vrcholů u v podstromu pod v takových, že $b(u \to v) = b$. Všimněte si, že v definici $b(u \to v)$ nejsme omezeni na podstrom v a naše trasa z něj může vylézt, abychom dobili. Každý potomek v bude v C_v započítán právě jednou, ledaže z něj do v vůbec nejde dojet.

Kromě počítání C_v také algoritmus při opouštění vrcholu v do globálního výsledku přičte všechny vyhovující dvojice a, b v podstromu pod v takové, že a i b zároveň neleží v podstromu stejného syna vrcholu v. Tak zaručíme, že každá vyhovující dvojice bude v celém algoritmu započítána právě jednou, a to v okamžiku, kdy se její vrcholy poprvé ocitnou ve stejném podstromu.

Popišme nyní samotný algoritmus. Jsme ve vrcholu v, který má syny $s_1, \dots, s_\ell$, $\ell \geq 0$, jejichž pole C_{s_i} jsou již spočítaná. Na začátku položíme $C_v \leftarrow [0, 0, \dots, 0, 1]$. Budeme postupně syny procházet a slévat jejich pole C_{s_i} s C_v, za chvíli řekneme jak. Pole C_v nám průběžně bude počítat, z kolika vrcholů z prvních $i-1$ podstromů umíme dorazit do v s tím kterým nabitím baterie. Souběžně se sléváním budeme pro každého syna s_i do globálního výsledku započítávat trasy, které mají jeden vrchol

v podstromu pod s_i a druhý vrchol buď v podstromu pod s_j, $j < i$, nebo ve v. Rozmyslete si, že takto během celého algoritmu z každého páru vyhovujících dvojic a, b a b, a započítáme právě jednu.

Nyní už samotný technický popis. Popíšeme zpracování syna s_i, máme-li už zpracované všechny předchozí syny. Vytvoříme pole $C^{\uparrow}_{s_i}[0, \ldots, k] = [0, \ldots, 0]$, kde $C^{\uparrow}_{s_i}[b]$ počítá počet vrcholů u pod s_i, které mají $b(u \to v) = b$ (tedy rozdíl mezi C_{s_i} a $C^{\uparrow}_{s_i}$ je ten, že jedno počítá s $b(u \to s_i)$ a druhé už s $b(u \to v)$). Nyní chceme pomocí C_{s_i} spočítat $C^{\uparrow}_{s_i}$. Pro každé nabití $b \in \{1, \ldots, k\}$ postupně položíme $x \leftarrow C_{s_i}[b]$. Chceme zjistit, jakého nejlepšího nabití ve v umíme pro tyto vrcholy (kterých je x) dosáhnout. Nejprve položíme $b' = b - 1$ (projdeme po hraně do v) a, stejně jako v kvadratickém řešení, je-li $d(v) \leq b' \leq k - d(v)$, položíme $b' = k - d(v)$ (stavíme se v nejbližší stanici a vrátíme se do v). Poté přičteme x k $C^{\uparrow}_{s_i}[b']$.

Teď už stačí k C_v popřičítat hodnoty z $C^{\uparrow}_{s_i}$, ještě předtím ale započítáme vyhovující trasy. Z pozorování o obrácení směru cestování plyne, že pro každý vrchol u s $b(u \to v) = b$ chceme započítat přesně ty vrcholy w s $b(w \to v) \geq k - b$. Konkrétně, pro každý vrchol započítaný v $C^{\uparrow}_{s_i}[b]$ chceme započítat právě

$$\sum_{j=k-b}^{k} C_v[j]$$

tras (kde zápis $\sum_{i=L}^{R} \xi(i)$ je zkratka za $\xi(L) + \xi(L+1) + \cdots + \xi(R)$). Celkově tedy za celého syna s_i započítáme

$$\sum_{b=0}^{k} \left(C^{\uparrow}_{s_i}[b] \cdot \sum_{j=k-b}^{k} C_v[j] \right)$$

tras. Tato dvojitá suma sčítá celkem $\mathcal{O}(k^2)$ členů, ale můžeme ji spočítat v čase $\mathcal{O}(k)$, pokud si všimneme, že vnitřní sumu umíme rychle přepočítávat (nebo si pro ni můžeme předpočítat prefixové součty).

Po přičtení tras už jen provedeme $C_v[b] \leftarrow C_v[b] + C^{\uparrow}_{s_i}[b]$ pro všechna b a pokračujeme s dalším synem. Všimněte si, že tento popis pokrývá i okrajové případy, a rozmyslete si, co se stane, když v nemá žádné syny, nebo když je ve v dobíjecí stanice.

Tím je popis algoritmu hotov. Algoritmus takto postupně provádí slévání polí C_v a průběžně do globálního výsledku počítá vyhovující dvojice, které mají za nejnižšího společného předchůdce zrovna aktuální vrchol. Po doběhnutí je v globálním výsledku počet všech vyhovujících dvojic, který ještě vynásobíme dvěma.

Jaká je časová složitost algoritmu? Může se zdát, že až $\Theta(n^2 k)$, protože každý vrchol může mít $\mathcal{O}(n)$ synů a ve vrcholu s ℓ syny strávíme $\mathcal{O}(k\ell)$ času. Budeme-li však čas účtovat tak, že práci, kterou provedeme při slévání pole syna s otcovým polem, naúčtujeme synům, strávíme ve skutečnosti v každém vrcholu dohromady jen $\mathcal{O}(k)$ času ($\mathcal{O}(k)$ přímo a dalších $\mathcal{O}(k)$ později při slévání). Celková časová složitost tedy je $\mathcal{O}(nk)$.

Na závěr poznamenejme, že úloha jde řešit i v čase $\mathcal{O}(n \log n)$. Algoritmus je trikový a používá tzv. *centroidovou dekompozici*, což je specifická metoda pracující na principu rozděl a panuj. Ve zkratce, vybereme si nějaký vrchol zhruba „uprostřed“ stromu a započítáme všechny vyhovující dvojice, které by po smazání centrálního vrcholu ležely v jiných podstromech. Poté vrchol pomyslně odstraníme a zavoláme se rekurzivně na každý menší podstrom, který takto vznikne. Budeme-li centrální vrchol volit šikovně, umíme zaručit, že se po jeho odstranění strom rozpadne na dostatečně malé části, a rekurze tak bude mít malou hloubku a časová složitost vyjde $\mathcal{O}(n \log n)$. Detailní popis tohoto řešení je však nad rámec našeho textu.

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

int n, k;
vector<bool> stanice;
vector<vector<int>> e;
vector<int> d; // Vzdálenost od dobíjecí stanice
vector<vector<int>> C;
long long res; // Globální proměnná pro počítání výsledku

void compute_d () { // BFS ze všech vrcholů s dobíjecími stanicemi
    queue<int> q;
    d.resize(n, n + 1);
    for (int i = 0; i < n; ++i) if (stanice[i]) {
        q.push(i);
        d[i] = 0;
    }
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (auto v : e[u]) {
            if (d[v] == n + 1) {
                d[v] = d[u] + 1;
                q.push(v);
            }
        }
    }
}

// Zpracuje vrchol v (do nějž přišlo z rodiče parent)
void solve_vertex (int v, int parent = -1) {
    // Navštívíme potomky
    for (auto u : e[v]) if (u != parent)
        solve_vertex(u, v);

    for (auto u : e[v]) if (u != parent) {
        // Zpracujeme syna u
        vector<int> Cx(k + 1, 0);
        for (int i = 1; i <= k; ++i) {
            int b = i - 1;
            if (b >= d[v]) b = max(b, k - d[v]);
            Cx[b] += C[u][i];
```

```
        }

        // Teď započítáme cesty přes v
        long long pref_sum = 0;
        for (int i = 0; i <= k; ++i) {
            pref_sum += C[v][k - i];
            res += Cx[i] * pref_sum;
        }

        for (int i = 0; i <= k; ++i) {
            C[v][i] += Cx[i];
        }
    }
}

int main () {
    int s;

    cin >> n >> k >> s;

    stanice.resize(n, false);
    e.resize(n);

    int a, b;
    for (int i = 0; i < s; ++i) {
        cin >> a;
        stanice[a - 1] = true; // Číslujeme od nuly
    }

    for (int i = 0; i < n - 1; ++i) {
        cin >> a >> b;
        --a; --b; // Číslujeme od nuly
        e[a].push_back(b);
        e[b].push_back(a);
    }

    compute_d();

    C.resize(n);
    for (int i = 0; i < n; ++i) {
        C[i].resize(k + 1, 0);
        C[i][k] = 1;
    }

    res = 0;
    solve_vertex(0);
    cout << res * 2 << endl;
}
```

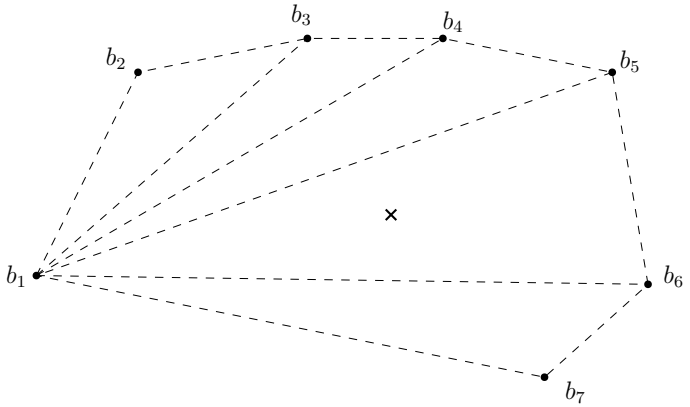
P-III-2 Základny na Marsu

Tato úloha byla geometrická, budeme tu proto používat některé geometrické pojmy a algoritmy, o nichž si můžete přečíst v kuchařce KSP (*https://ksp.mff.cuni.cz/kucharky/geometrie/*). Konkrétně budeme využívat konvexní obal, což je nejmenší konvexní mnohoúhelník, uvnitř nějž leží všechny zadané body, a fakt, že pomocí determinantů resp. vektorových součinů umíme v konstantním čase určit, na které straně orientované přímky leží zadaný bod. Algoritmus na konstrukci konvexního obalu n bodů běží v čase $\mathcal{O}(n \log n)$.

Začněme pozorováním, že pro tři vysílače a jednu základnu umíme v konstantním čase zjistit, zda základna leží uvnitř trojúhelníka vymezeného vysílači – to nastane právě tehdy, když základna leží na stejné straně všech tří orientovaných přímek určených vysílači v nějakém pevném cyklickém uspořádání. Z toho rovnou dostáváme řešení za dva body, v čase $\mathcal{O}(m^3 n)$ stačí projít všechny trojice vysílačů, pro každou z nich projít všechny základny a pro každou z nich ověřit, zda leží uvnitř trojúhelníka. Jelikož máme slíbené, že vysílače jsou v obecné poloze a žádná základna neleží na přímce určené vysílači, nebudeme muset řešit žádné speciální případy.

Konvexní obal

Abychom mohli řešení zrychlit, uvědomme si nejprve, že základna leží uvnitř nějakého trojúhelníku z vysílačů, právě tehdy když leží uvnitř konvexního obalu všech vysílačů. Implikace doprava je zřejmá, předpokládejme tedy, že základna leží uvnitř konvexního obalu všech vysílačů. Vyberme si libovolný vrchol konvexního obalu a spojme ho úsečkami se všemi ostatními vrcholy konvexního obalu. Tím jsme obal rozdělili na po dvou disjunktní trojúhelníky, a pokud základna ležela uvnitř obalu, musí nyní ležet uvnitř nějakého trojúhelníku (viz následující obrázek).



Všimněte si, že v předchozím odstavci jsme dokázali něco silnějšího, totiž že pokud základna leží uvnitř konvexního obalu všech vysílačů, tak pro každý vysílač ležící na hranici konvexního obalu umíme najít další dva vysílače takové, aby dohromady tvořily trojúhelník, uvnitř něj bude základna ležet. Na řešení za pět bodů nám tedy stačí na začátku najít jeden vysílač, který určitě bude ležet na hranici konvexního obalu, a pak v čase $\mathcal{O}(m^2 n)$ pro každý vysílač projít všechny dvojice základen a provést kontrolu stejně jako u řešení za dva body. Najít takový vysílač je jednoduché, stačí vzít například vysílač, který má nejmenší hodnotu souřadnice x (a pokud jsou dva takové, tak libovolný z nich). Z algoritmu na hledání konvexního obalu v kuchařce KSP plyne, že takový vysílač vždy bude ležet na konvexním obalu.

Sedm bodů bylo možné získat v podstatě za implementaci pozorování o konvexním obalu. V čase $\mathcal{O}(m \log m)$ najdeme konvexní obal vysílačů, uspořádáme si vrcholy cyklicky podle toho, jak na konvexním obalu leží (např. podle hodinových ručiček), vybereme si jeden vrchol obalu a obal s jeho pomocí rozdělíme na $\mathcal{O}(m)$

trojúhelníků. Pro každou základnu nám pak stačí projít všechny tyto trojúhelníky a ověřit, zda základna v jednom z nich leží. Takové řešení běží v čase $\mathcal{O}(mn+m\log m)$.

Binární vyhledávání

Pro zisk plného počtu bodů však ani takové řešení nestačí, nemůžeme si dovolit pro každou základnu procházet až $\mathcal{O}(m)$ trojúhelníků. Pro zjednodušení nyní předpokládejme, že vrcholy konvexního obalu máme uspořádané po směru hodinových ručiček jako $b_1, b_2, \ldots, b_\ell$. Tehdy platí, že pokud základna leží uvnitř trojúhelníka b_1, b_i, b_{i+1} pro nějaké $1 < i < \ell$, pak leží napravo od (orientovaných) přímek b_1b_j pro všechna $1 < j \leq i$ a nalevo od přímek b_1b_j pro všechna $i < j \leq \ell$. To znamená, že můžeme pro každou základnu v čase $\mathcal{O}(\log m)$ binárně vyhledat takové i, že základna je napravo od přímky b_1b_i a nalevo od přímky b_1b_{i+1} (nebo zjistit, že neexistuje, v takovém případě je odpověď pro danou základu -1) a ověřit, zda základna leží uvnitř trojúhelníka $b_1b_ib_{i+1}$. Takové řešení běží v čase $\mathcal{O}((m+n)\log m)$ a mohlo získat až deset bodů. Paměťová složitost byla u všech řešení lineární.

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

using namespace std;

typedef long long ll;

struct point {
    int id;
    ll x;
    ll y;

    bool operator== (const point& b) {
        return id == b.id && x == b.x && y == b.y;
    }

    bool operator< (const point& b) {
        // Třídíme lexikograficky
        if (x == b.x) return y < b.y;
        return x < b.x;
    }
};

int m, n;
vector<point> vysilace;
vector<point> obal;

// Vrátí true, pokud body a, b, c jsou v tomto pořadí ve směru hodinových ručiček
// (tj. bod c je napravo od orientované přímky ab)
bool clockwise (point a, point b, point c) {
    ll ux = c.x - a.x;
    ll uy = c.y - a.y;
    ll vx = b.x - a.x;
    ll vy = b.y - a.y;

    return ux*vy - uy*vx > 0;
}

// Po zavolání této funkce budou v poli vysilace pouze body ležící na konvexním
// obalu, uspořádané cyklicky po směru hodinových ručiček
```

```
vector<point> create_convex_hull (vector<point> input) {
    sort(input.begin(), input.end());
    vector<point> horni_polovina, dolni_polovina;

    for (auto p : input) {
        while (horni_polovina.size() >= 2 &&
            !clockwise(horni_polovina[horni_polovina.size()-2],
                       horni_polovina[horni_polovina.size()-1], p)) {
            horni_polovina.pop_back();
        }
        horni_polovina.push_back(p);
    }

    for (int i = input.size() - 1; i >= 0; --i) {
        auto p = input[i];
        while (dolni_polovina.size() >= 2 &&
            !clockwise(dolni_polovina[dolni_polovina.size()-2],
                       dolni_polovina[dolni_polovina.size()-1], p)) {
            dolni_polovina.pop_back();
        }
        dolni_polovina.push_back(p);
    }

    vector<point> output = horni_polovina;
    for (int i = 1; i + 1 < dolni_polovina.size(); ++i)
        output.push_back(dolni_polovina[i]);

    return output;
}

void vyres_stanici (point stanice) {
    int l = 1, r = obal.size() - 1;
    while (r - l > 1) {
        int m = (l + r) / 2;
        // cerr << m << endl;
        if (clockwise(obal[0], obal[m], stanice)) l = m;
        else r = m;
    }

    // Pokud leží stanice v konvexním obalu, pak leží v trojúhelníků s vrcholy
    // obal[0], obal[l], obal[r]

    if (clockwise(obal[0], obal[l], stanice) &&
        clockwise(obal[l], obal[r], stanice) &&
        clockwise(obal[r], obal[0], stanice))
        cout << obal[0].id << " " << obal[l].id << " " << obal[r].id << endl;
    else
        cout << "-1" << endl;
}

int main () {
    cin >> m >> n;

    ll x, y;
    for (int i = 1; i <= m; ++i) {
        cin >> x >> y;
        vysilace.push_back(point{i, x, y});
    }

    obal = create_convex_hull(vysilace);
```

```
    for (auto p : obal) {
        cout << p.id << ":   " << p.x << " " << p.y << endl;
    }

    for (int i = 0; i < n; ++i) {
        point stanice;
        cin >> stanice.x >> stanice.y;
        vyres_stanici(stanice);
    }
}
```

P-III-3 Hledání na disku

Podúloha a) – binární vyhledávání

Binární vyhledávání v posloupnosti N prvků provede v nejhorším případě $\log N$ kroků, kde log značí dvojkový logaritmus. Číslujeme-li kroky od 0, pak v i-tém kroku hledáme v úseku délky $N/2^i$. (Pokud prostřední prvek nezapočítáme ani do menších, ani do větších, zmenšují se úseky trochu rychleji, ale ne natolik, aby to ovlivnilo asymptotiku.)

Algoritmus v každém kroku přečte jeden prostřední prvek úseku (říkejme mu *střed*). Dokud jsou úseky velké, leží každý střed v jiném bloku na disku. Přesněji řečeno: Má-li úsek v i-tém kroku aspoň $2B$ prvků, leží jeho střed v jiném bloku než střed $(i+1)$-tého úseku. Takže dokud je $N/2^i \geq 2B$, každý další krok musí načíst nový blok. Tato nerovnost je ekvivalentní s $N/B \geq 2^{i+1}$, po zlogaritmování $i \leq \log(N/B) - 1$.

Naopak pokud už je úsek krátký, tedy $N/2^i \leq 2B$, vejde se celý do tří bloků. Jakmile tyto bloky načteme do vnitřní paměti, nepotřebujeme dál číst z disku. Celý zbytek výpočtu proto nic dalšího nečte.

Dokázali jsme tedy, že komunikační složitost binárního vyhledávání je rovna $\log(N/B) + \mathcal{O}(1)$ neboli $\log N - \log B + \mathcal{O}(1)$.

Podúloha b) – efektivní vyhledávání

Další známou datovou strukturou pro hledání prvků v množině jsou binární vyhledávací stromy. Ty můžeme na disk uložit po hladinách (podobně jako haldu). Podobná úvaha jako v předchozí části by nicméně vedla k závěru, že během hledání jednoho prvku potřebujeme přečíst až $\log N - \log B$ prvků. (Zde je naopak „levná“ počáteční část, kdy se pohybujeme blízko kořene, takže vrcholy stromu jsou v paměti blízko sebe.)

Abychom dosáhli nižší komunikační složitosti, budeme potřebovat z jednoho přečteného bloku vytěžit více informací než výsledek jednoho porovnání.

Binární strom proto zobecníme na *vícecestný vyhledávací strom* (takovým stromům se někdy říká (a,b)-stromy nebo B-stromy). Definujeme ho následovně. Každý vrchol stromu bude mít nějaký obecný počet synů a bude v něm uložen jeden nebo více klíčů ve vzestupném pořadí. Přitom obsahuje-li vrchol nějaké klíče $x_1 < \ldots < x_k$, má $k+1$ synů. V binárním stromu platilo, že klíč v libovolném vrcholu je větší než všechny klíče v levém podstromu a menší než všechny v pravém. Zde bude platit, že

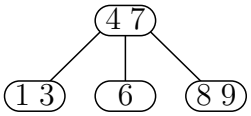
první podstrom obsahuje klíče menší než x_1, druhý ty mezi x_1 a x_2, …, až $(k+1)$-tý klíče větší než x_k. V listech už jsou uloženy jen klíče.

Hledání probíhá podobně jako v binárním stromu: kdykoliv přijdeme do vrcholu, porovnáme hledaný prvek se všemi klíči a podle toho vybereme podstrom, v němž budeme pokračovat.

Počty klíčů v jednotlivých vrcholech můžeme volit libovolně. My je zvolíme tak, aby se jeden vrchol vešel do jednoho bloku disku a co nejlépe ho využil. Každý vrchol tedy bude obsahovat B klíčů (ukazatele na syny nemusíme ukládat, podobně jako haldu můžeme i tento strom uložit na disk po hladinách a podle pozice otce spočítat pozice synů; kdybychom přesto chtěli ukazatele ukládat, místo B synů jich budeme mít $B/2$, což asymptotiku neovlivní).

Strom budeme stavět následovně (detaily uložení na disku neřešíme, zadání to po nás nechce). Množinu setřídíme a rozdělíme ji na $B+1$ úseků. Hraniční prvky úseků se stanou klíči v kořeni stromu, z každého úseku se stane jeden podstrom. Podstromy vytvoříme rekurzivní aplikací téhož algoritmu. Listy stromu už mohou obsahovat méně než B klíčů.

Strom může vypadat třeba takto:



Každý krok konstrukce zmenší vstup $(B+1)$-krát, takže po $\log_{B+1} N \approx \log_B N$ krocích se dostaneme do listů. Strom má tedy výšku $\log_B N$.

Hledání proto projde $\log_B N$ vrcholů a v každém z nich načte jeden blok. Komunikační složitost tedy činí $\mathcal{O}(\log_B N) = \mathcal{O}(\log N / \log B)$.

Ještě ověříme, že jsme nepokazili časovou složitost. Projdeme $\mathcal{O}(\log N / \log B)$ vrcholů, v každém potřebujeme zjistit, mezi které dva klíče patří hledaný prvek. To můžeme vyhledat binárně (tentokrát v rámci jediného bloku) v čase $\mathcal{O}(\log B)$. Časová složitost bez načítání bloků tedy činí $\mathcal{O}(\log N / \log B \cdot \log B) = \mathcal{O}(\log N)$. To je stále optimální.

Podúloha c) – důkaz optimality

Nejprve si rozmyslíme, že libovolný algoritmus pro vyhledávání musí provést alespoň $\log N$ porovnání v nejhorším případě. Budeme uvažovat jen deterministické algoritmy (nepoužíváme žádnou náhodu) a prvky budeme mít dovoleno pouze porovnávat (žádné indexování pole hodnotou prvku – to nám zadání zakazuje).

Možné průběhy algoritmu můžeme popsat stromem. V každém vnitřním vrcholu stromu se algoritmus zeptá na porovnání hledaného prvku s nějakým prvkem množiny (na porovnání dvou prvků množiny se nemá smysl ptát, když je množina statická). Vnitřní vrchol má dva syny, které odpovídají tomu, jak algoritmus pokračuje, pokud je hledaný prvek větší nebo menší. Výpočet skončí, pokud nastane rovnost nebo pokud se dostaneme do listu stromu. Odpověď zvolíme podle toho, ve kterém vrcholu výpočet skončil.

Algoritmus ovšem musí umět dát N různých výsledků, takže strom musí mít aspoň N vrcholů. Nicméně binární strom výšky h má nejvýše $2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$ vrcholů. Pro výšku stromu tedy musí platit $2^{h+1} - 1 \geq N$, tedy $h + 1 \geq \log N$. Alespoň jedna cesta mezi kořenem a listem tedy má alespoň $\log N$ vrcholů. Proto pro alespoň jeden vstup algoritmus provede alespoň $\log N$ porovnání.

Nyní budeme místo provedených porovnání počítat bloky načtené z disku. Jeden vrchol stromu tedy odpovídá načtení jednoho bloku. Jeho synové odpovídají možným vztahům hledaného prvku k prvkům uloženým v bloku. Jelikož prvky můžeme jenom porovnávat, každý takový vztah je určen tím, kam by se do setříděné posloupnosti prvků v bloku zařadil hledaný prvek. To dává nejvýše $(2B+1)$ možností – B prvků a $B+1$ „děr“ mezi nimi. Kdykoliv porovnání skončí rovností, algoritmus skončí. Takže vrchol má nejvýše $B + 1$ synů.

Nyní má strom výšky h nejvýše $(B+1)^h$ listů a nejvýše tolik vnitřních vrcholů. V každém vrcholu může rovnost nastat nejvýše B způsoby, takže možných odpovědí algoritmu je maximálně $2B \cdot (B+1)^h \leq (B+1)^{h+2}$. Toto číslo musí být opět větší nebo rovno N, takže $h + 2 \geq \log_{B+1} N$. Z toho plyne, že $h \geq c \log_B N$ pro nějakou konstantu c.

Takže každý algoritmus musí v nejhorším případě načíst aspoň $c \log_B N$ bloků. Proto je náš algoritmus z části b) optimální až na multiplikativní konstantu.