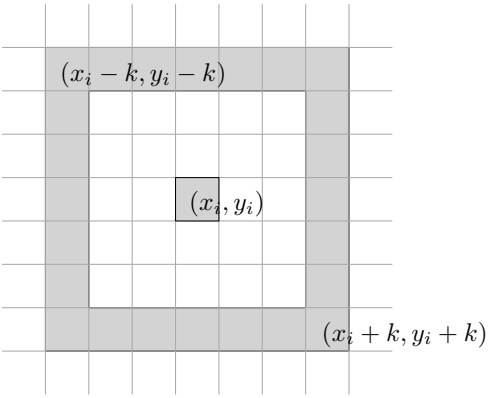
P-II-1 Let na Mars

Tři body bylo možné získat za řešení hrubou silou, které pro každou dvojici $1 \leq i < j \leq n$ vyzkouší, zda splňuje požadované podmínky, a vypíše počet těch dvojic, které podmínky splňovaly. Časová složitost takového řešení je $\mathcal{O}(n^2)$ a paměťová $\mathcal{O}(n)$.

Ve vstupech za více bodů bylo n příliš veliké na to, abychom měli čas iterovat přes všechny dvojice. Pořiďme si nyní veliký čtverečkovaný papír a pro každé i umisťme kosmonauta číslo i na políčko se souřadnicemi (x_i, y_i). Podívejme se na kosmonauta i a všimněme si, že kosmonaut j může s i letět právě tehdy, když nastane jedna z následujících čtyř možností:

- $x_j = x_i + k$ a $|y_j - y_i| \leq k$,
- $x_j = x_i - k$ a $|y_j - y_i| \leq k$,
- $y_j = y_i + k$ a $|x_j - x_i| \leq k$,
- $y_j = y_i - k$ a $|x_j - x_i| \leq k$.

Jinak řečeno, kosmonauti, kteří mohou s kosmonautem číslo i letět na Mars, jsou právě ti, kteří stojí na obvodu „čtverce“ tlustém jedno políčko, který obsahuje políčka $(x_i - k, y_i - k)$ a $(x_i + k, y_i + k)$. Situace pro $k = 3$ je znázorněna na následujícím obrázku:



Prefixové součty

Kdybychom v konstantním čase uměli spočítat počet kosmonautů v libovolném takto zadaném čtverci, mohli bychom potom sečíst tyto výsledky pro každého kosmonauta a vydělit je dvěma (jelikož každou dvojici započítáme dvakrát), čímž bychom získali výsledek. Toto je možné udělat pomocí dvourozměrných prefixových součtů –

pokud je ještě neznáte, pěkné povídání o nich najdete v KSP kuchařce v kapitole Základní algoritmy (*https://ksp.mff.cuni.cz/kucharky/zakladni-algoritmy/*). Použijeme je následovně. Označme X jako maximum ze všech x_i, Y jako maximum za všech y_i a pořiďme si dvojrozměrné pole A velikosti $(X+1) \times (Y+1)$, kde

$$A[x][y] = \begin{cases} 1 & \text{pokud existuje kosmonaut } i, \text{ pro nějž } (x,y) = (x_i, y_i), \\ 0 & \text{jinak.} \end{cases}$$

To lze udělat s časovou i paměťovou složitostí $\mathcal{O}(n + XY)$. V čase $\mathcal{O}(XY)$ si pak předpočítáme dvourozměrné prefixové součty pole A podle KSP kuchařky, které nám umožní v čase $\mathcal{O}(1)$ zjistit počet kosmonautů v libovolném obdélníku. Počet kosmonautů na obvodu našeho čtverce pak lze získat jako rozdíl počtů ve vnějším resp. vnitřním čtverci, opět v konstantním čase. To celé nám tedy dává časovou i paměťovou složitost $\mathcal{O}(n + XY)$, což stačilo na 6 bodů.

Optimální řešení

Na zisk plného počtu bodů žádné prefixové součty potřebovat nebudeme. Místo toho si všimněme, že pokud si profily kosmonautů setřídíme primárně podle první souřadnice a sekundárně podle druhé, tak pro každé i tvoří profily kosmonautů j splňující $x_j = x_i + k$ a $|y_j - y_i| \leq k$, respektive $x_j = x_i - k$ a $|y_j - y_i| \leq k$ v tomto poli souvislé úseky. Analogicky, pokud si profily setřídíme primárně podlé druhé souřadnice a sekundárně podle první, tak pro každé i tvoří profily kosmonautů j splňující $y_j = y_i + k$ a $|x_j - x_i| \leq k$, respektive $y_j = y_i - k$ a $|x_j - x_i| \leq k$ v tomto poli souvislé úseky.

Předpokládejme, že pro setříděné pole X velikosti n, v němž se neopakují hodnoty, umíme v čase $\mathcal{O}(\log n)$ zjistit počet prvků X, které leží (neostře) mezi danými $u \leq v$ (tj. počet $x \in X$ takových, že $u \leq x \leq v$). Ukážeme si, jak s takovým předpokladem vyřešit úlohu v čase $\mathcal{O}(n \log n)$. Nejprve si vytvoříme pole P a D obsahující všechny profily setříděné primárně podle první a sekundárně podle druhé souřadnice (pole P), resp. primárně podle druhé a sekundárně podle první souřadnice (pole D). To jistě dokážeme v čase $\mathcal{O}(n \log n)$.

Potom pro každé i od 1 do n zjistíme, kolik prvků je v P neostře mezi prvky $(x_i - k, y_i - k)$ a $(x_i - k, y_i + k)$, respektive $(x_i + k, y_i - k)$ a $(x_i + k, y_i + k)$. Stejně tak zjistíme, kolik prvků je v D neostře mezi prvky $(x_i - k, y_i - k)$ a $(x_i + k, y_i - k)$, respektive $(x_i - k, y_i + k)$ a $(x_i + k, y_i + k)$, a všechna tato čísla sečteme. To téměř odpovídá počtu kosmonautů, kteří s kosmonautem i mohou letět na Mars, až na to, že pokud nějaký kosmonaut má profil $(x_i \pm k, y_i \pm k)$ (ty odpovídají čtyřem rohům čtverce), tak jsme ho započetli dvakrát. Pro každou ze čtyř možností můžeme v čase $\mathcal{O}(\log n)$ zjistit, zda kosmonaut s takovým profilem existuje (dotazem na počet prvků neostře mezi $(x_i - k, y_i - k)$ a $(x_i - k, y_i - k)$ a analogicky pro zbylé tři možnosti), a pokud ano, tak od celkového počtu odečíst jedna.

Takto spočítáme počet uspořádaných dvojic kosmonautů i, j, kteří spolu mohou letět na Mars. Abychom dostali odpověď na úlohu, musíme toto číslo vydělit dvěma. Celý algoritmus poběží s časovou složitostí $\mathcal{O}(n \log n)$ a paměťovou složitostí $\mathcal{O}(n)$.

Počet prvků v intervalu

Zbývá tedy vysvětlit, jak v setříděném poli X velikosti n (indexovaném od 0), které neobsahuje opakující se hodnoty, najít pro libovolné $u \leq v$ v čase $\mathcal{O}(\log n)$ počet prvků $x \in X$, které splňují $u \leq x \leq v$. Všimněme si, že pokud i je index nejmenšího prvku, který je větší nebo roven u (případně n, pokud žádný takový neexistuje), a pokud j je index nejmenšího prvku, který je ostře větší než v (případně n, pokud žádný takový neexistuje), pak $j - i$ dává odpověď na naši otázku. Jak i, tak j lze nalézt pomocí binárního vyhledávání v čase $\mathcal{O}(\log n)$. Ukážeme si to pro i, pro nalezení j můžeme zavolat stejný algoritmus s v místo u, a pokud je hodnota pole na vráceném indexu rovna v, zvýšit index o 1 (tady využíváme toho, že se v X neopakují hodnoty). Pokud jste nikdy pojem binární vyhledávání neslyšeli, doporučujeme příslušnou kapitolu KSP kuchařky (*https://ksp.mff.cuni.cz/kucharky/binarni-vyhledavani/*).

Pokud $X[0] \geq u$, pak vrátíme 0. Jinak si pořídíme dvě proměnné ℓ a r a na začátku nastavíme $\ell \leftarrow 0$ a $r \leftarrow n$. Dokud $r - \ell > 1$, budeme provádět následující kroky:

1. $m \leftarrow \lfloor (r + \ell)/2 \rfloor$ ◁ *celočíselné dělení*
2. Pokud $X[m] < u$, nastavíme $\ell \leftarrow m$.
3. Jinak nastavíme $r \leftarrow m$.

Nakonec vrátíme r.

Všimněme si, že v každém kroku platí, že $X[\ell] < u$ a buď $r = n$, nebo $X[r] \geq u$. To znamená, že ve chvíli, kdy $r - \ell = 1$, ukazuje r na nejmenší prvek větší nebo roven u (případně $r = n$), což je to, co hledáme. Jako obvykle se při analýze binárního vyhledávání ukáže, že v každém kroku se rozdíl $r - \ell$ zmenší zhruba na polovinu, tedy uděláme nejvýše $\mathcal{O}(\log n)$ kroků (opět viz KSP kuchařka, *https://ksp.mff.cuni.cz/kucharky/binarni-vyhledavani/*).

Poznámky k implementaci

Kontrolu, zda jsme některý z prvků $(x_i \pm k, y_i \pm k)$ započítali dvakrát, nemusíme dělat zvlášť, lze k tomu využít nalezené indexy z podúlohy „počet prvků v intervalu“. V ukázkové implementaci jsme v poli Y prohodili první a druhou souřadnici, díky čemuž jsme ho mohli opět řadit primárně podle první a sekundárně podle druhé souřadnice. Alternativní řešení by bylo používat vlastní porovnávací funkce, což má v C++ ne úplně triviální syntaxi.

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

using namespace std;

// Vrátí index nejmenšího prvku pole vec, který je >= u
int nejmensi_vetsi(vector<pair<int, int>> vec, pair<int, int> u) {
    if (vec[0] >= u) return 0;
    int l = 0;
```

```
    int r = 1 + vec.size();
    while (r - l > 1) {
        int m = (l + r) / 2;
        if (vec[m] < u) l = m;
        else r = m;
    }
    return r;
}

int main() {
    int n, k;
    vector<pair<int, int>> X, Y;

    cin >> n >> k;

    X.resize(n);
    Y.resize(n);

    for (int i = 0; i < n; ++i) {
        cin >> X[i].first >> X[i].second;
        Y[i].first = X[i].second;
        Y[i].second = X[i].first; // V poli Y prohodíme první a druhou souřadnici
    }

    sort(X.begin(), X.end());
    sort(Y.begin(), Y.end());
    // Díky prohození souřadnic můžeme Y také řadit lexikograficky

    long long pocet_dvojic = 0;

    for (int kosmonaut = 0; kosmonaut < n; ++kosmonaut) {
        pair<int, int> profil = X[kosmonaut];
        pair<int, int> u, v;

        // Potřebujeme spočítat počet kosmonautů na čtyřech stranách čtverce,
        // každou uděláme zvlášť
        int znamenka[] = {-1, 1};

        // Nejprve to uděláme pro svislé strany čtverce a pole X
        for (int znam : znamenka) {
            // Strany čtverce
            // (x_kosmonaut+-k, y_kosmonaut-k)...(x_kosmonaut+-k, y_kosmonaut+k)
            u.first = profil.first + znam*k;
            u.second = profil.second - k;
            v.first = profil.first + znam*k;
            v.second = profil.second + k;

            int i = nejmensi_vetsi(X, u);
            int j = nejmensi_vetsi(X, v);

            // (x_kosmonaut+-k, y_kosmonaut-k) bychom započítali dvakrát
            if (X[i] == u) pocet_dvojic--;
            // (x_kosmonaut+-k, y_kosmonaut+k) bychom započítali dvakrát
            if (X[j] == v) pocet_dvojic--;

            if (j < n && X[j] == v) j++; // Chceme X[j] > v nebo j = n

            pocet_dvojic += j - i;
        }

        // Totéž pro vodorovné strany a pole Y
        for (int znam : znamenka) {
```

```
            // Strany čtverce
            // (x_kosmonaut-k, y_kosmonaut+-k)...(x_kosmonaut+k, y_kosmonaut+-k)

            // Musíme prohodit souřadnice
            u.first = profil.second + znam*k;
            u.second = profil.first - k;
            v.first = profil.second + znam*k;
            v.second = profil.first + k;

            int i = nejmensi_vetsi(Y, u);
            int j = nejmensi_vetsi(Y, v);

            if (j < n && Y[j] == v) j++; // Chceme Y[j] > v nebo j = n

            pocet_dvojic += j - i;
        }
    }

    cout << (pocet_dvojic / 2) << endl;
}
```

P-II-2 Laboratorní protokoly

Definujme si novou posloupnost C takovou, že $C_i = |A_i - B_i|$ pro každé $1 \leq i \leq n$. Všimněme si, že $\sum_{i=1}^{n}(A_i - B_i)^2 = \sum_{i=1}^{n} C_i^2$ a že přičtení/odečtení jedničky od B_i odpovídá přičtení/odečtení jedničky od C_i (přičtení nebo odečtení podle toho, zda $A_i \geq B_i$ nebo ne). Jelikož nás zajímá pouze minimální hodnota výrazu $\sum_{i=1}^{n}(A_i - B_i)^2$, můžeme místo posloupností A a B pracovat rovnou s posloupností C.

Tím jsme úlohu v čase $\mathcal{O}(n)$ převedli na její jednodušší variantu: Na vstupu je posloupnost C délky n, která obsahuje *nezáporná* celá čísla, a nezáporné celé číslo k. Úkolem je nejvýše k-krát přičíst/odečíst jedničku od prvků C tak, abychom minimalizovali výraz $\sum_{i=1}^{n} C_i^2$. Dále vyřešíme tuto úlohu.

Nejprve si všimněme, že pokud $k \geq \sum_{i=1}^{n} C_i$, tak můžeme všechny hodnoty C_i vynulovat a tím na nulu dostat i odchylku, což je zřejmě optimální řešení. Toto lze ověřit v čase $\mathcal{O}(n)$ a odteď můžeme předpokládat, že $k < \sum_{i=1}^{n} C_i$.

Hladové řešení

Nejdříve dokážeme, že úloha lze řešit hladově, tj. že správné řešení lze získat algoritmem, který bude mít k fází, v každé fázi najde i takové, že C_i je (neostře) největší prvek C a sníží jej o 1. (Jelikož jsme předpokládali, že $k < \sum_{i=1}^{n} C_i$, nalezený největší prvek C_i bude vždy splňovat $C_i > 0$.) Po těchto k fázích algoritmus vypíše $\sum_{i=1}^{n} C_i^2$.

Uvažujme nyní nějakou posloupnost D, kterou lze z C získat po nejvýše k přičteních/odečteních jedničky, takovou, že $\sum_{i=1}^{n} D_i^2$ je optimální, tedy nejmenší možné. Označme jako $X_1, \ldots, X_n$ sestupně seřazené prvky posloupnosti, kterou z C po k krocích vytvoří náš algoritmus, a jako $Y_1, \ldots, Y_n$ sestupně seřazené prvky posloupnosti D. Dokážeme, že $X_i = Y_i$ pro všechna i.

Z definice našeho algoritmu vyplývá, že prvky, které upravoval, tvoří počáteční úsek posloupnosti X, a navíc platí, že rozdíl každých dvou upravených prvků je nejvýše 1. Pro spor předpokládejme, že existuje pozice i taková, že $X_i \neq Y_i$, a uvažujme nejmenší takové i. Jelikož náš algoritmus jedničky pouze odečítal a oba algoritmy

udělaly celkem k kroků, musí platit $X_i < Y_i$. Navíc musí existovat $j > i$, pro nějž $X_j > Y_j$. To speciálně znamená, že optimální řešení od prvku Y_j alespoň jednou odečetlo jedničku.

Uvažme nyní řešení, které oproti optimálnímu o jedna méněkrát odečetlo jedničku a naopak od Y_i odečetlo jedničku o jedna vícekrát. Jeho optimalizovaný výraz má hodnotu

$$Y_1^2 + \cdots + Y_{i-1}^2 + (Y_i - 1)^2 + Y_{i+1}^2 + \cdots + Y_{j-1}^2 + (Y_j + 1)^2 + Y_{j+1}^2 + \cdots + Y_n^2,$$

což lze upravit na

$$\left(\sum_{i=1}^{n} Y_i^2\right) - 2Y_i + 1 + 2Y_j + 1 = \left(\sum_{i=1}^{n} Y_i^2\right) + 2(Y_i - Y_j + 1).$$

Jelikož $Y_i > X_i \geq X_j > Y_j$, máme $Y_i - Y_j \geq 2$, a tedy $2(Y_j - Y_j + 1) < 0$. To znamená, že nové řešení dosáhlo ještě menší hodnoty optimalizovaného výrazu, což je spor s existencí pozice i, kde $X_i \neq Y_i$. Takže $X = Y$ a náš algoritmus vrátí optimální hodnotu, což jsme chtěli dokázat.

Všimněme si, že jsme ve skutečnosti dokázali o něco více: Pokud spustíme náš hladový algoritmus na posloupnost C a tu poté seřadíme sestupně, bude platit, že náš algoritmus upravil právě prvky nějakého počátečního úseku C, každé dva upravené prvky se liší nejvíce o jedna a navíc je takový počáteční úsek nejkratší možný (mezi všemi takovými seřazenými posloupnostmi, které lze získat po nejvýše k validních úpravách). Část „a navíc“ plyne z toho, že kdyby tomu tak nebylo, můžeme uvažovanou posloupnost s delším upraveným počátečním úsekem použít jako D v důkazu správnosti a dostat tak spor.

Implementace

Dalším krokem je implementovat hladový algoritmus tak, aby běžel v čase $\mathcal{O}(n \log n)$. Přímá implementace algoritmu by běžela v čase $\mathcal{O}(kn)$, jelikož v každém z k kroků musíme projít celou posloupnost C a najít její největší prvek. Pokud bychom prvky C uložili do (maximové) haldy nebo binárního vyhledávacího stromu, zlepšila by se složitost na $\mathcal{O}((k + n) \log n)$ — v každém kroku najdeme maximum, to odebereme, snížíme o jedna a vrátíme do haldy/vyhledávacího stromu, to celé v čase $\mathcal{O}(\log n)$

Jelikož časová složitost optimální implementace nemá být závislá na k, je potřeba vymyslet, jak dělat více kroků najednou. K tomu využijeme pozorování, že setřídíme-li si výsledné hodnoty po proběhnutí algoritmu sestupně, tak platí, že algoritmus upravoval pouze hodnoty, které se vyskytují v počátečním úseku, tyto hodnoty se liší nejvýše o jedna a navíc je takový počáteční úsek nejkratší možný.

Optimální implementace rovnou nalezne takto setříděnou posloupnost. Začneme tím, že nějakým standardním algoritmem na třídění (např. mergesort) setřídíme posloupnost C sestupně v čase $\mathcal{O}(n \log n)$, a z technických důvodů na chvíli přidáme $C_{n+1} = 0$. (Což zachová, že je posloupnost setříděná.)

Nyní chceme najít pozici i takovou, že náš hladový algoritmus by upravil prvních i prvků setříděné posloupnosti C. Víme, že hladový algoritmus celkem zmenší prvních i prvků posloupnosti o k a že i poté zůstane posloupnost seřazená. Jinak řečeno, každý z prvních i prvků musí i po zmenšení být větší nebo roven C_{i+1}. To znamená, že musí platit $k \leq S - i \cdot C_{i+1}$, kde S je součet prvních i prvků.

Pořídíme si proto proměnné i a S, které nastavíme $i = 0$ a $S = 0$, a dokud $k > S - i \cdot C_{i+1}$, nastavíme $i := i + 1$ a $S := S + C_i$ (v tomto pořadí). Všimněte si, že se tento cyklus zastaví nejpozději pro $i = n$, jelikož $C_{n+1} = 0$ a my předpokládáme, že $k < \sum_{i=1}^{n} C_i$. Nyní můžeme opět odstranit C_{n+1}. Tato část zřejmě poběží v čase $\mathcal{O}(n)$.

Index i je pozice posledního prvku posloupnosti, který budeme zmenšovat. Chceme zmenšit právě i prvků, jejichž počáteční součet je S, celkem je musíme zmenšit přesně o k a na konci se žádné dva nesmí lišit o více než 1. Jinak řečeno, část z nich snížíme na nějaké ℓ a zbytek na $\ell - 1$.

To znamená, že $k = S - i \cdot \ell + r$, kde r je počet prvků, které snížíme na $\ell - 1$, a platí $0 \leq r < i$. Tuto rovnici můžeme přeuspořádat na $\ell = \frac{S-k+r}{i}$ a z toho vyjádříme, že $\ell = \lceil \frac{S-k}{i} \rceil$, kde $\lceil$ a $\rceil$ značí horní celou část, a $r = k - S + i \cdot \ell$. Jelikož $k \leq S - i \cdot C_{i+1}$, dosazením dostaneme $S - i \cdot \ell + r \leq S - i \cdot C_{i+1}$, tedy $C_{i+1} + r/i \leq \ell$.

Nyní nastavíme prvních $i - r$ prvků C na ℓ a dalších r prvků C na $\ell - 1$ (to lze udělat v čase $\mathcal{O}(n)$). Pokud $r = 0$, nastavili jsme všech i prvků na ℓ, a protože $C_{i+1} \leq \ell$, posloupnost zůstala seřazená sestupně. Jinak $r > 0$, tedy i $r/i > 0$, a proto $\ell > C_{i+1}$. Jelikož ℓ i C_{i+1} jsou celá čísla, dostaneme, že ve skutečnosti $\ell - 1 \geq C_{i+1}$, tedy opět posloupnost C zůstala seřazená sestupně.

Nakonec zbývá vypsat hodnotu $\sum_{i=1}^{n} C_i$. Časová složitost řešení je $\mathcal{O}(n \log n)$, paměťová složitost je $\mathcal{O}(n)$, což stačilo na plný počet bodů.

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

using namespace std;

int main() {
    int n;
    int k;
    cin >> n >> k;
    vector<long long> A, B, C;
    A.resize(n);
    B.resize(n);     // A a B indexujeme pro jednoduchost od 0
    C.resize(n+1);   // C budeme indexovat od 1, aby fungovaly vzorečky

    for (int i = 0; i < n; ++i) cin >> A[i];
    for (int i = 0; i < n; ++i) cin >> B[i];

    long long sm = 0;

    for (int i = 1; i <= n; ++i) {
        C[i] = A[i-1] - B[i-1];
        if(C[i] < 0) C[i] *= -1;
        sm += C[i];
```

```
    }

    if (sm <= k) {  // V tomto případě můžeme celou posloupnost vynulovat
        cout << "0" << endl;
        return 0;
    }

    // ++C.begin() ukazuje na prvek s indexem 1, tj. ignorujeme C[0]
    sort(++C.begin(), C.end(), std::greater<long long>());

    C.push_back(0);
    int i = 0;
    long long S = 0;

    while (k > S - i * C[i+1]) {
        ++i;
        S += C[i];
    }

    long long l =  (S - k + i - 1) / i; // Horní celá část (S - k) / i
    int r = k - S + i * l;

    int j = 1;
    for (; j <= i - r; ++j) C[j] = l;
    for (; j <= i; ++j) C[j] = l - 1;

    long long res = 0;

    for (int i = 1; i <= n; ++i) res += C[i] * C[i];

    cout << res << endl;
    return 0;
}
```

P-II-3 Vánoční návštěvy

V řešení budeme používat standardní terminologii a základní algoritmy teorie grafů. Pokud je neznáte, doporučujeme například KSP kuchařku o grafech dostupnou na stránce *https://ksp.mff.cuni.cz/kucharky/*. V této terminologii dostaneme na vstupu konstantu k a neorientovaný graf se třemi typy vrcholů (salámisté/nesalámisté/nikdo). Pro zjednodušení budeme vrcholy označovat jako vrcholy typu S (salámisté), N (nesalámisté) a 0 (nikdo).

Úloha nám dává za úkol zjistit, zda i po smazání libovolného vrcholu typu S a všech hran s ním incidentních (to odpovídá návštěvě tetičky Gertrudy, která znemožní projet daným vrcholem) existuje posloupnost $v_1, \ldots, v_s$ vrcholů G splňující následující podmínky:

- Pro každé $1 \leq i < s$ platí, že vrcholy v_i a v_{i+1} jsou spojeny hranou.
- $v_1 = 1$ a $v_s = n$. *(Začneme ve vrcholu* 1 *a skončíme ve vrcholu n.)*
- Pokud pro nějaké i je v_i typu S, pak pro každé $j > i$ platí, že v_j je typu 0. *(Po návštěvě salámistické vesnice již nesmíme navštívit žádné příbuzné.)*
- Pokud pro nějaké i platí, že všechny vrcholy $v_i, v_{i+1}, \ldots, v_{i+k-1}$ jsou typu N, pak vrchol v_{i+k} je typu 0. *(Salát se smí jíst nejvýše k krát za sebou.)*

Poznamenejme, že posloupnost splňující první podmínku se nazývá *sled* a od pojmu *cesta* použitého v domácím kole se liší tím, že ve sledu se mohou opakovat

vrcholy i hrany. Sled splňující zadané podmínky budeme nazývat jako *vyhovující sled.*

Kdybychom předpokládali, že salámisté bydlí v nejvýše 10 vesnicích, můžeme postupně každou z nich zkusit smazat (tím simulujeme, že si tetička Gertruda pro návštěvu vybrala právě ji) a zjistit, zda ve vzniklém grafu existuje sled splňující zadané podmínky. To se řeší analogicky jako v úloze Přívalový déšť z domácího kola s předpokladem $mk \leq 10^6$, totiž vytvoříme si nový orientovaný graf, který z původního získáme přidáním dvou informací navíc: Kolik vrcholů s příbuznými v kuse již Novákovi navštívili a zda již někdy navštívili salámistický vrchol. Orientované hrany se stejně jako v domácím kole definují tak, aby odpovídaly simulaci jízdy Novákových. Tento graf budeme nazývat *nafouknutý graf.*

Poté stačí spustit prohledávání do hloubky nebo do šířky z vrcholu 1 a stavu „v kuse Novákovi navštívili 0 vrcholů s příbuznými“ a „zatím nenavštívili žádný salámistický vrchol“ a zjistit, zda během prohledávání dojdeme do vrcholu n s libovolným stavem. Pokud je odpověď ano pro každý možný smazaný salámistický vrchol, vypíšeme `ANO`, jinak vypíšeme `NE`. Nafouknutý graf bude mít nejvýše $\mathcal{O}(nk)$ vrcholů a $\mathcal{O}(mk)$ hran, jeho vytvoření i prohledání tudíž poběží v čase $\mathcal{O}(k(n+m))$. Jelikož máme nejvýše 10 vrcholů typu S, toto celé spustíme nejvýše desetkrát. Takové řešení mohlo získat až pět bodů.

Chceme-li získat plný počet bodů, musíme se vyhnout odebírání vrcholů typu S po jednom. Všimněme si, že libovolný vyhovující sled navštíví maximálně jednou vrchol typu S. Pokud najdeme vyhovující sled, který nenavštíví žádný vrchol typu S, máme vyhráno (jelikož Novákovi vždy mohou využít tento sled bez ohledu na to, kam přijede tetička Gertruda). To lze ověřit jedním prohledáním (do šířky nebo hloubky) nafouknutého grafu v čase $\mathcal{O}(k(n+m))$, při němž budeme ignorovat salámistické vrcholy. Dále budeme předpokládat, že takový sled neexistuje.

V tom případě je odpověď `ANO` právě tehdy, když existují alespoň dva vyhovující sledy využívající různé vrcholy typu S. Každý takový sled lze rozdělit na úsek z vrcholu 1 do vrcholu typu S a na zbylou část z vrcholu typu S do vrcholu n. Důležité pozorování je, že druhá část využívá (až na počáteční vrchol typu S) pouze vrcholy typu 0. My budeme hledat tyto sledy po částech. Nejprve najdeme všechny vrcholy typu S, do kterých se dá dostat z vrcholu n přes vrcholy typu 0, a potom najdeme všechny vrcholy typu S, do kterých se dá dostat z vrcholu 1 se splněním všech pravidel. Odpověď na úlohu je `ANO`, právě tehdy když tyto dvě množiny mají alespoň dvouprvkový průnik.

Na nalezení vrcholů typu S dosažitelných z vrcholu n pomocí vrcholů typu 0 stačí obyčejné prohledání (do šířky nebo hloubky) vstupního grafu z vrcholu n, které nepoužívá vrcholy typu N a ve vrcholech typu S zastavuje. Standardní implementace poběží v čase $\mathcal{O}(n+m)$. Pro nalezení vrcholů typu S dosažitelných z vrcholu 1 při splnění všech pravidel si vytvoříme nafouknutý graf a spustíme jeho prohledávání z vrcholu 1 a stavu „v kuse Novákovi navštívili 0 vrcholů s příbuznými“ a „zatím nenavštívili žádný salámistický vrchol“ a podíváme se, které vrcholy typu S v jakémkoliv stavu navštíví. To má časovou i paměťovou složitost $\mathcal{O}(k(n+m))$. Nakonec

v čase $\mathcal{O}(n)$ projdeme všechny vrcholy a spočítáme, kolik z nich má typ S a zároveň jsme je navštívili během obou prohledávání. Pokud budou alespoň dva, vypíšeme `ANO`, jinak vypíšeme `NE`.

Celková časová i paměťová složitost řešení je tedy $\mathcal{O}(k(n+m))$, což stačilo pro získání 10 bodů.

Na závěr poznamenáme, že při implementaci lze udělat několik zjednodušení. Jednak můžeme hledat vyhovující sled z vrcholu 1 do n bez použití vrcholů typu S společně s hledáním vrcholů typu S dosažitelných z vrcholu 1, jelikož při tomto prohledávání nemusíme z vrcholů typu S pokračovat dále, jednak ve skutečnosti nemusíme graf nafukovat i o stavy, zda již byl navštíven nějaký vrchol typu S – při prohledávání v nich zastavujeme.

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

using namespace std;

int n, m, k;
vector<int> typ;        // Vrcholy budeme číslovat 0...(n-1) misto 1...n
vector<vector<int>> e;  // Hrany

// Prohledávání do šířky z vrcholu n-1 procházející pouze vrcholy typu 0
// a hledající dosažitelné vrcholy všech typů
vector<bool> najdi_dosazitelne_vrcholy_z_chaty() {
    vector<bool> dosazitelny(n, false);
    queue<int> q;

    dosazitelny[n-1] = true;
    q.push(n - 1);

    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int w : e[v]) { // Iterujeme přes všechny sousedy v
            if (dosazitelny[w]) continue;
            dosazitelny[w] = true;
            if (typ[w] == 0) {
                q.push(w); // Procházíme pouze skrz vrcholy typu 0
            }
        }
    }
    return dosazitelny;
}

// Prohledávání do šířky v nafouknutém grafu z vrcholu 0 podle pravidel.
// Ze salámistických vrcholů dál nepokračujeme.
// Vrátí, které vrcholy původního grafu jsou v nějakém stavu dosažitelné.
vector<bool> najdi_dosazitelne_vrcholy_z_domu() {
    vector<vector<bool>> navstiveny;
    // První souřadnice je číslo vrcholu, druhá je počet salátů v řadě
    navstiveny.resize(n);
    for (int i = 0; i < n; ++i)
        navstiveny[i].resize(k + 1, false);
```

```
    vector<bool> dosazitelny(n, false);
    queue<pair<int, int>> q;
    // První prvek je číslo vrcholu, druhý je počet salátů v řadě
    // (včetně příslušného vrcholu)

    navstiveny[0][0] = true;
    q.push({0, 0});

    while (!q.empty()) {
        pair<int, int> v = q.front();
        q.pop();
        dosazitelny[v.first] = true;
        if (typ[v.first] == 2) continue; // Od salámistů nepokračujeme

        for (int w : e[v.first]) {
            // Po k salátech je potřeba odpočinek
            if (v.second >= k && typ[w] != 0) continue;

            // Nový počet salátů v řadě
            int nova_sytost;
            if (typ[w] == 1 || typ[w] == 2) nova_sytost = v.second+1;
            else nova_sytost = 0;

            // Pokud jsme v novém stavu již byli, ignorujeme jej
            if (navstiveny[w][nova_sytost]) continue;

            navstiveny[w][nova_sytost]=true;
            q.push({w, nova_sytost});
        }
    }

    return dosazitelny;
}

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

    typ.resize(n);
    e.resize(n);

    for (int i = 0; i < n; ++i) cin >> typ[i];

    int x, y;
    for (int i = 0; i < m; ++i) {
        cin >> x >> y;
        --x; --y; // Číslujeme od nuly
        e[x].push_back(y);
        e[y].push_back(x);
    }

    vector<bool> dosazitelne_z_chaty = najdi_dosazitelne_vrcholy_z_chaty();
    vector<bool> dosazitelne_z_domu = najdi_dosazitelne_vrcholy_z_domu();

    // Pokud se na chatu umíme dostat bez návštěvy salámistů, máme vyhráno
    if (dosazitelne_z_domu[n-1]) {
        cout << "ANO" << endl;
        return 0;
    }

    // Jinak vyhrajeme, právě když se z chaty i domu umíme dostat
    // do nějakých dvou vrcholů typu S
    int pocet_spolecnych_salamistu = 0;
```

```
    for (int i = 0; i < n - 1; ++i) {
        if (typ[i] == 2 && dosazitelne_z_chaty[i] && dosazitelne_z_domu[i])
            ++pocet_spolecnych_salamistu;
    }

    if (pocet_spolecnych_salamistu >= 2)
        cout << "ANO" << endl;
    else
        cout << "NE" << endl;
}
```

P-II-4 Třídění na disku

Podúloha a) – třídění výběrem

Minimum pokaždé hledáme z nejvýše n po sobě jdoucích prvků na disku, takže stačí přečíst nejvýše n/B po sobě jdoucích bloků. Prohození prvků provedeme přečtením příslušných bloků z disku do paměti, prohozením v paměti a zapsáním zpět. To k n/B přístupům na disk přidá pouze konstantu.

Tuto operaci provádíme n-krát, takže celkem mezi pamětí a diskem přeneseme $\mathcal{O}(n^2/B)$ bloků.

Časová složitost vyjde stejně jako u třídění výběrem v paměti: jeden výběr minima trvá $\mathcal{O}(n)$, všech n výběrů tedy $\mathcal{O}(n^2)$.

Podúloha b) – efektivní algoritmus

Efektivněji lze třídit *Mergesortem* neboli algoritmem *třídění slévánim.*

Je založený na slévání dvou setříděných posloupností do jedné. Jak na disku provést samotné slévání, uvidíme vzápětí. Předtím ukážeme, jak z něj vybudovat třídění.

Představíme si, že se celá posloupnost skládá ze setříděných úseků (těm se obvykle říká *běhy*) nějaké délky u. To je na počátku jistě pravda: jednotlivé prvky tvoří triviální běhy délky $u = 1$. V každém průchodu třídění budeme slévat dvojice sousedních běhů a vytvářet běhy délky $2u$.

Po prvním průchodu tedy máme běhy délky 2, po druhém délky 4, ..., po p-tém průchodu běhy délky 2^p. Pokud je délka posloupnosti n mocnina dvojky, po $\log_2 n$ průchodech se celá posloupnost skládá z jediného běhu délky n, takže je setříděná.

Pokud n není mocnina dvojky, představíme si, že posloupnost doplníme hodnotami $+\infty$ na délku nejbližší vyšší mocniny dvojky. To složitost algoritmu zhorší nejvyše konstanta-krát. (Dokonce si můžeme nekonečna pouze představovat a operace s nimi ve skutečnosti neprovádět.)

Víme tedy, že nám stačí $\mathcal{O}(\log n)$ průchodů. Nyní doplníme technické detaily.

Popíšeme algoritmus pro slévání dvou setříděných posloupností $x_1, \ldots, x_r$ a $y_1, \ldots, y_s$ do posloupnosti $z_1, \ldots, z_{r+s}$. Bude fungovat takto: nejprve porovná x_1 s y_1 a menší z nich „odtrhne“ a přesune do z_1. Poté pokračuje slévánim toho, co z posloupností zbylo. Stačí si tedy pamatovat aktuální prvek v každé vstupní posloupnosti a v každém kroku přesunout do výstupní posloupnosti menší z aktuálních prvků:

```
1.  i ← 1, j ← 1, k ← 1          ◁ aktuální jsou x_i a y_j, vytváříme z_k
2.  Dokud i ≤ r a j ≤ s:
3.      Je-li x_i ≤ y_j:                    ◁ menší je v posloupnosti x
4.          z_k ← x_i, i ← i + 1
5.      Jinak:                                          ◁ menší je v y
6.          z_k ← y_j, j ← j + 1
7.      k ← k + 1
8.  Pokud i < r:      ◁ jedna posl. skončila, zkopírujeme zbytek druhé
9.      z_k, ..., z_{r+s} ← x_i, ..., x_r
10. Jinak:
11.     z_k, ..., z_{r+s} ← y_j, ..., y_s
```

Algoritmus vytvoří $r + s$ prvků a každým stráví konstantní čas, takže celkem běží v čase $\mathcal{O}(r + s)$, tedy lineárním v celkovém počtu prvků.

Snadno ho můžeme použít i pro data na disku: stačí nám jeden sekvenční průchod přes posloupnost x, druhý přes y a třetí přes výsledek z. Celkem přeneseme $\mathcal{O}(\lceil r/B \rceil + \lceil s/B \rceil)$ bloků a postačí nám 3 bloky paměti: po jednom na aktuální pozici v každé ze tří posloupností. (Plus samozřejmě potřebujeme někde skladovat řídicí proměnné, ale zadání slibuje, že na to paměť vždy máme.)

Jeden průchod Mergesortu slévání postupně aplikuje na $n/(2u)$ dvojic běhů délky u. Ty všechny čte ze zdrojové posloupnosti a výsledky zapisuje za sebe do cílové posloupnosti. V dalším průchodu si zdrojová a cílová posloupnost prohodí role. Všechna slévání v rámci průchodu trvají dohromady $\mathcal{O}(n)$ času, protože dohromady přesunula n prvků.

Podobně je to s přístupy na disk. Podívejme se na první běhy ve dvojicích: když je běh slévání čteme, procházíme celý vstup od začátku do konce, jen při tom přeskakujeme druhé běhy dvojic. Celkem tedy přečteme nejvýše n/B bloků. Podobně všechna čtení druhých běhů dohromady přečtou nejvýše n/B bloků. A vytváření výsledku zapisuje na disk sekvenčně, takže zapíše také n/B bloků. To je dohromady $\mathcal{O}(n/B)$ bloků. Co se vnitřní paměti týče, v každém okamžiku v ní stačí mít uložené 3 bloky, se kterými právě pracujeme.

Celkem tedy provedeme $\mathcal{O}(\log n)$ průchodů a každý z nich stráví čas $\mathcal{O}(n)$ a přenese $\mathcal{O}(n/B)$ bloků. To dohromady dává časovou složitost $\mathcal{O}(n \log n)$ a komunikační složitost $\mathcal{O}((n/B) \log n)$.

Podúloha b) – lepší řešení

Předchozí řešení se ještě dá vylepšit – zatím jsme totiž využili z vnitřní paměti pouze konstantní počet bloků.

Popišme obecně situaci, kterou jsme potkali při slévání. Zpracováváme nějakých q posloupností. Každou z nich procházíme sekvenčně, ale kdykoliv se můžeme zastavit a chvíli se zabývat jinou posloupností. Máme-li k dispozici q bloků paměti, můžeme si dovolit udržovat v paměti aktuální blok každé posloupnosti. Proto komunikační složitost vyjde stejná, jako kdybychom každou posloupnost četli zvlášť.

Jelikož naše paměť má M/B bloků, můžeme si takto dovolit zpracovávat až řádově M/B posloupností najednou.

To vede k myšlence vícecestného Mergesortu. Zavedeme si parametr t, který nám bude říkat, kolik posloupností sléváme najednou. Jeho konkrétní hodnotu zvolíme později. V jednom průchodu se nám tedy délka běhu neznásobí dvakrát, nýbrž t-krát. Po p-tém průchodu budou mít běhy délku t^p, takže po $\lceil \log_t n \rceil$ průchodech bude setříděno.

Zbývá nahlédnout, jak efektivně slévat t posloupností najednou. Myšlenka bude stejná jako pro slévání dvou, jen v každém kroku budeme hledat minimum z t aktuálních prvků. Nechceme ovšem, aby to trvalo čas lineární s t, a proto si pořidíme haldu velikosti t, pomocí které pokaždé najdeme a odebereme minimum z aktuálních prvků a nahradíme ho novým aktuálním prvkem. Jeden krok slévání tedy bude trvat $\mathcal{O}(\log t)$ času. Všechna slévání v rámci průchodu tedy potrvají $\mathcal{O}(n \log t)$.

Jak to provedeme na disku? Na jedno slévání budeme potřebovat $t + 1$ bloků paměti, po jednom na aktuální blok každé z t vstupních posloupností a jeden na aktuální blok posloupnosti výstupní. Pak také budeme potřebovat řádově t buněk paměti na uložení haldy, ale to je řádově méně paměti než t bloků. Vše dohromady se tedy určitě vejde do $2t$ bloků. Pak opět funguje úvaha, že každou posloupnost nezávisle na ostatních projdeme sekvenčně.

Trochu složitější je zkombinovat všechna slití, která nastanou během jednoho průchodu. Úvaha o přeskakování bloku z dvojcestného Mergesortu by se pro $t > 2$ zkomplikovala, tak to uděláme trochu jinak. Místo abychom slévali sousední běhy, tak rozdělíme běhy do t přibližně stejně velkých skupin. Pak slijeme všechny první běhy každé skupiny, pak všechny druhé a tak dále. V rámci každé skupiny tedy čteme sekvenčně a výsledek zase zapisujeme sekvenčně, takže postačí $\mathcal{O}(n/B)$ přístupů na disk.

V celém Mergesortu provádíme $\mathcal{O}(\log_t n)$ průchodů, každý stojí $\mathcal{O}(n \log t)$ času a $\mathcal{O}(n/B)$ přístupů na disk. Z toho vyjde časová složitost $\mathcal{O}(n \log t \log_t n)$. Jelikož $\log_t n = \log n / \log t$, můžeme to také psát jako $\mathcal{O}(n \log t \cdot (\log n / \log t)) = \mathcal{O}(n \log n)$. Komunikační složitost činí $\mathcal{O}((n/B) \log_t n)$.

Zbývá zvolit t. Z rozboru slévání plyne, že pro dané t spotřebujeme nejvýše $2t$ bloků paměti. Proto si můžeme dovolit nastavit $t = M/(2B)$. Časová složitost pak vyjde $\mathcal{O}(n \log n)$ a komunikační $\mathcal{O}((n/B) \log_{M/2B} n) = \mathcal{O}((n/B) \log_{M/B} n)$.

Nakonec dodejme, že je známo, že tato komunikační složitost je nejlepší možná. Důkaz ale není snadný.