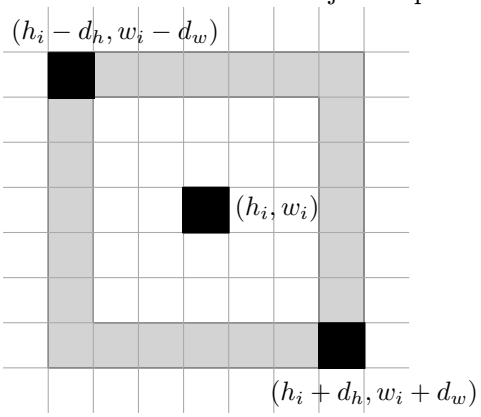
A-II-1 Kozmonauti

Tri body sa dali získať za priamočiare riešenie hrubou silou: pre každú dvojicu dobrovoľníkov vyhodnotíme, či spĺňajú podmienky zo zadania. (Podotýkame ale, že na tri body bolo potrebné splniť všetky formálne náležitosti – uviesť dôležitú časť implementácie, popis riešenia a odhad jeho časovej zložitosti.)

Prefixové súčty

Ak máme malé M, môžeme si celý vstup nakresliť ako bitmapu: výšku a váhu každého dobrovoľníka budeme chápať ako súradnice políčka v nej.
Keď máme takúto bitmapu, vedeli by sme úlohu riešiť v čase $O(nM)$ jednoducho tak, že pre každého dobrovoľníka prejdeme cez všetky kombinácie výšky a váhy, ktoré môže mať jeho partner, a zakaždým sa pozrieme do bitmapy, či taký partner existuje.
Odhad časovej zložitosti vyplýva z toho, že ak poznáme parametre h_i a w_i prvého dobrovoľníka, tak druhý dobrovoľník musí spĺňať buď ($h_j = h_i \pm d_h$ a $w_j \in [w_i - d_w, w_i + d_w]$) alebo naopak ($h_j \in [h_i - d_h, h_i + d_h]$ a $w_j = w_i \pm d_w$). Každej možnosti zodpovedá nanajvýš zhruba $4M$ prípustných kombinácií výšky a váhy druhého dobrovoľníka.
Takéto riešenie ešte nestačí na šesť bodov, vieme ho ale zefektívniť vhodným predpočítaním. Potrebujeme vlastne vedieť počet dobrovoľníkov na obvode obdĺžnika v našej bitmape.



Na toto môžeme ľahko použiť tzv. prefixové súčty. Môžeme si predstaviť, že na každom políčku našej bitmapy máme napísané *číslo*: počet dobrovoľníkov s presne takými parametrami. Potom odpoveď, ktorú hľadáme pre konkrétneho dobrovoľníka, je súčtom čísel vo všetkých na obrázku vyznačených políčkach.
Existovali dva zhruba rovnocenné prístupy. Pri prvom si predpočítame prefixové súčty zvlášť pre každý riadok a pre každý stĺpec. Potom ľubovoľný súčet, ktorý hľadáme, získame ako súčet vhodných dvoch kusov riadkov a dvoch kusov stĺpcov. Druhou možnosťou bolo rovno použiť dvojrozmerné prefixové súčty a odpoveď získať ako súčet celého obdĺžnika mínus súčet jeho vnútra.
Detaily o tejto technike nájdete v Kuchárke KSP: https://www.ksp.sk/kucharka/prefixove_sumy/ a https://www.ksp.sk/kucharka/2d_prefixove_sumy/.

Vzorové riešenie

Predchádzajúce riešenie upravíme tak, aby sme pracovali len s tými políčkami bitmapy, ktoré naozaj niekoho obsahujú.
Predstavme si teda každého dobrovoľníka ako jedno políčko so súradnicami (h_i, w_i). Všimnime si, čo sa stane, keď všetkých dobrovoľníkov usporiadame podľa výšky a pri rovnakej výške podľa váhy. Povedzme teraz, že hľadáme dobrovoľníkovi i partnera spĺňajúceho $h_j = h_i - d_h$. Kľúčovým pozorovaním je, že v našom usporiadanom poli tvoria takíto dobrovoľníci *súvislý úsek*. A keďže majú všetci rovnakú výšku, sú usporiadaní podľa hmotnosti, a teda dokonca platí, že tí z nich, ktorí sú naozaj kompatibilní s dobrovoľníkom i, tvoria súvislý úsek v našom poli.
Takýto úsek vieme efektívne nájsť v čase $O(\log n)$ pomocou binárneho vyhľadávania. (V programe to robíme tak, že knižničnými funkciami `lower_bound` a `upper_bound` nájdeme iterátory na začiatok a koniec tohto úseku. V

konštantnom čase potom vieme od seba tieto iterátory odčítať a zistiť, aký dlhý úsek medzi nimi leží.)
Následne vieme rovnako nájsť úsek vyhovujúcich partnerov s výškou $h_i + d_h$. No a navyše si pripravíme aj *druhé* usporiadané pole, tiež obsahujúce všetkých dobrovoľníkov. Toto pole si usporiadame primárne podľa váhy a až sekundárne podľa výšky. V takomto poli zase platí, že keď poznáme presnú váhu človeka, ktorého hľadáme, dostávame jeden súvislý úsek kandidátov usporiadaných podľa výšky.
A tým sme už takmer hotoví, len si treba dať pozor na to, aby sme nezapočítali dvakrát ľudí, ktorí sa od aktuálneho dobrovoľníka líšia aj vo výške o presne d_h aj vo váhe o presne d_w. V programe to robíme tak, že pri presnej váhe sa pozeráme len na výšky v intervale od $h_i - d_h + 1$ po $h_i + d_h - 1$.
Takéto riešenie má časovú zložitosť $O(n \log n)$: aj kvôli triedeniu, aj kvôli tomu, že potom pre každého dobrovoľníka robíme konštantne veľa binárnych vyhľadávaní, každé v čase $O(\log n)$. Pamäťová zložitosť je zjavne lineárna.

Listing programu (C++)

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

// struktura na ulozenie dobrovolnika
struct dobrovolnik { int h, w; };

// porovnavacie funkcie primarne podla vysky a primarne podla vahy
bool podla_vysky(const dobrovolnik &A, const dobrovolnik &B) {
    if (A.h != B.h) return A.h < B.h; else return A.w < B.w;
}

bool podla_vahy(const dobrovolnik &A, const dobrovolnik &B) {
    if (A.w != B.w) return A.w < B.w; else return A.h < B.h;
}

int main() {
    // nacitame vstup
    int N, dh, dw;
    cin >> N >> dh >> dw;
    vector<dobrovolnik> DHW, DWH;
    for (int n=0; n<N; ++n) {
        int h, w;
        cin >> h >> w;
        DHW.push_back( {h,w} );
        DWH.push_back( {h,w} );
    }

    // usporiadame obe postupnosti
    sort( DHW.begin(), DHW.end(), podla_vysky );
    sort( DWH.begin(), DWH.end(), podla_vahy );

    // pre kazdeho dobrovolnika zistime pocet kompatibilnych
    long long odpoved = 0;
    for (int i=0; i<N; ++i) {
        int h = DHW[i].h, w = DHW[i].w;
        // zaratame tych s extremnou vyskou
        for (int hj : {h-dh, h+dh}) {
            auto lo = lower_bound( DHW.begin(), DHW.end(), dobrovolnik{hj,w-dw}, podla_vysky );
            auto hi = upper_bound( DHW.begin(), DHW.end(), dobrovolnik{hj,w+dw}, podla_vysky );
            odpoved += (hi - lo);
        }
        // zaratame tych s extremnou vahou ale nie extremnou vyskou
        for (int wj : {w-dw, w+dw}) {
            auto lo = lower_bound( DWH.begin(), DWH.end(), dobrovolnik{h-dh+1,wj}, podla_vahy );
            auto hi = upper_bound( DWH.begin(), DWH.end(), dobrovolnik{h+dh-1,wj}, podla_vahy );
            odpoved += (hi - lo);
        }
    }

    // kazdu dvojicu sme zaratali dvakrat
    cout << (odpoved / 2) << endl;
}
```

A-II-2 Laboratórne protokoly

Ak sa Mišova a Rišova hodnota líšia, je zjavne jedno, či jednu zväčšíme alebo druhú zmenšíme, ich rozdiel sa tým zmení rovnako. Stačí sa preto pozerať len na tieto rozdiely. Máme teda postupnosť $a_i = |m_i - r_i|$ a chceme minimalizovať súčet všetkých a_i^2.

Vzorové riešenie zostrojíme postupne v dvoch krokoch. Najskôr si dokážeme, že pri riešení tejto úlohy funguje jednoduchá pažravá stratégia, a potom si ukážeme, ako ju efektívne implementovať.

Pažravý algoritmus

Pažravá stratégia je pomerne zjavná. V prvom rade je zjavné, že chceme hodnoty a_i postupne v nejakom poradí znižovať, až kým buď nedostaneme všetky na nulu alebo nespravíme všetkých k povolených úprav.
V druhom rade si môžeme všimnúť, že ak zmenšíme 1 na 0, zmenšíme tým odchýlku pre toto meranie len o 1, ale ak namiesto toho napríklad zmenšíme iný prvok 10 na 9, zmenšíme jeho odchýlku z 10^2 na 9^2, teda až o 19. Zjavne platí, že čím väčší prvok postupnosti a zmenšíme, tým viac celková odchýlka klesne: zmena z a_i na $a_i - 1$ zmení odchýlku z a_i^2 na $(a_i - 1)^2$, čiže o $2a_i - 1$.
Toto pozorovanie nás teda privádza na myšlienku, že v každom kroku budeme chcieť zmenšiť aktuálne najväčšiu z hodnôt a_i. Toto tvrdenie si teraz dokážeme.
Zjavne stačí dokázať, že v prvom kroku riešenia môžeme zmenšiť najväčšiu z hodnôt a_i. Korektnosť celého algoritmu potom vyplynie jednoducho z opakovaného použitia tejto úvahy.
Nech teda bez ujmy na všeobecnosti $a_1 = \max(a) > 0$. Tvrdíme, že existuje optimálne riešenie, ktoré v prvom kroku zmenší a_1.
Majme ľubovoľne riešenie. Ak toto riešenie v ľubovoľnom kroku zmenší a_1, môže ho zmenšiť hneď v prvom kroku – na poradí krokov nezáleží. A ak toto riešenie nikdy hodnotu a_1 nezmenší, pozrime sa na to, čo spravilo v poslednom kroku: zmenšilo vtedy nejakú inú hodnotu. Tá hodnota vtedy nemohla byť väčšia ako a_1 na začiatku, a teda ak namiesto toho, že na konci zmenšíme ju, zmenšíme hneď na začiatku a_1, dostaneme aspoň rovnako dobré riešenie.
Ukázali sme teda, že ku každému riešeniu vieme nájsť aspoň rovnako dobré riešenie, ktoré začne tým, že zmenší najväčšiu hodnotu. Tým je naše tvrdenie dokázané.
Šikovná priamočiara implementácia tohto pažravého riešenia vedela získať 7 bodov. Prvky a_i si uložíme do vhodnej dátovej štruktúry, ktorá nám umožní efektívne hľadať aktuálne najväčší z nich. Asi najjednoduchšou takouto štruktúrou je binárna halda s maximom v koreni. No a potom už len k-krát vyberieme maximum, o 1 ho zmenšíme a vložíme ho naspäť. Celková časová zložitosť takéhoto riešenia vie byť $O(n + k \log n)$.

Efektívna implementácia

Predstavme si, že sme si všetky hodnoty a_i usporiadali od najväčšej po najmenšiu. Situáciu si môžeme znázorniť graficky, každé a_i bude jeden riadok guličiek. Napr. $a = (9, 7, 6, 6, 6, 3, 1)$ si predstavíme nasledovne:

```
OOOOOOOOO
OOOOOOO
OOOOOO
OOOOOO
OOOOOO
OOO
O
```

Keď teraz postupne zmenšujeme tieto a_i, na obrázku sa to prejaví tak, že sprava doľava postupne odoberáme guličky. V rámci stĺpca je jedno, v akom poradí to robíme, budeme to preto robiť systematicky zhora dole. Napr. vo vyššie znázornenej situácii by sme pre $k = 12$ postupne odobrali guličky, ktoré sú nižšie nahradené hviezdičkami.

```
OOOO*****
OOOO***
OOOO**
OOOOO*
OOOOO*
OOO
O
```

Tento proces chceme vedieť robiť efektívnejšie. Prvým pokusom by mohlo byť neodoberať guličky po jednej ale robiť to stĺpec po stĺpci. Toto ešte nestačí: najlepšie, čo vieme dosiahnuť, je, že namiesto k krokov ich budeme mať zhruba k/n, najhorší prípad ostáva dokonca rovnaký – ak je jedno a_i výrazne väčšie ako ostatné, naďalej odoberáme guličky po jednej.
Lepšie bude postupovať po celých blokoch stĺpcov. Blok stĺpcov bude súvislá sada stĺpcov rovnakej dĺžky. V našom prípade teda bloky vyzerajú nasledovne:

```
O|OO|OOO|O|OO
O|OO|OOO|O|
O|OO|OOO|  |
O|OO|OOO|  |
O|OO|OOO|  |
O|OO|   |  |
O|  |   |  |
```

Blokov stĺpcov je vždy najviac n, lebo ako ideme sprava doľava, každý blok má viac riadkov ako ten predchádzajúci.
No a v každom bloku tvoria guličky obdĺžnik. Preň vieme v konštantnom čase vypočítať, aký je veľký, a teda či ho ešte odoberieme celý. A keď prídeme k bloku, ktorý už celý odobratý nebude, v lineárnom čase vieme povedať, v akom stave skončí.
Najpomalšou časťou celého tohto riešenia je teda usporiadanie hodnôt a_i, na ktoré potrebujeme $O(n \log n)$ času. Zvyšok riešenia je už lineárny od n.

Listing programu (Python)

```
N, K = [ int(_) for _ in input().split() ]
M = [ int(_) for _ in input().split() ]
R = [ int(_) for _ in input().split() ]

A = [ abs(m-r) for m,r in zip(M,R) ]
A = list(reversed(sorted(A)))
A.append(0)
print(A)

if K >= sum(A):
    print(0)
    exit()

for n in range(1,N+1):
    if A[n] < A[n-1]:
        print(n)
        # prvých n riadkov tvorí ďalší blok
        sirka = A[n-1] - A[n]
        print(n, sirka, K)
        if n * sirka < K:
            # celý zoberieme a ešte pokračujeme
            K -= n * sirka
        else:
            # v tomto bloku končíme
            # zistíme, koľko celých stĺpcov zoberieme a koľko zoberieme z nasledujúceho
            cele_stlpce = K // n
            ostalo = A[n-1] - cele_stlpce
            kratsie = K % n
            # odoberieme všetky tieto guličky a sčítame odchýlky za aktuálny blok
            odpoved = kratsie * (ostalo-1)**2
            odpoved += (n-kratsie) * ostalo**2
            # a pripočítame zatiaľ nedotknuté riadky, ktoré ešte do aktuálneho bloku nepatria
            odpoved += sum( a**2 for a in A[n:] )
            print(odpoved)
            break
```

A-II-3 Babičky

Vyriešme najskôr jednoduchšiu úlohu: bez deda Jozefa. Zaujíma nás teda len to, či existuje nejaká cesta, počas ktorej nás neprekŕmia.

Stavy

Predstavme si, že sa pohybujeme po dedine. V ľubovoľnom okamihu tesne po návšteve nejakého domu závisia naše možnosti od troch parametrov:

- Číslo domu, pri ktorom práve stojíme (n možností).
- Či sme už boli u nejakej babičky (2 možnosti).
- Akú dlhú sériu navštívených rodinných domov práve máme ($k + 1$ možností).

Ak poznáme tieto údaje, vieme o ľubovoľnom pláne ďalšej cesty povedať, či je alebo nie je prípustný. Tejto sade údajov budeme preto hovoriť *náš stav*.
Stavov je zjavne $2n(k + 1)$. Nie všetky stavy sú skutočne dosiahnuteľné – ak sme už boli u nejakej babičky, nemohli sme potom byť v žiadnych rodinných domoch, a teda naša séria má dĺžku 0. Skutočne dosiahnuteľných stavov je len $n + n(k + 1)$. Pri písaní programu je ale pohodlnejšie, ak všetky stavy „vyzerajú rovnako“, preto sme ich definovali vyššie uvedeným spôsobom.

Stavový priestor

Ak vieme, v akom sme stave, a pohneme sa k susednému domu, vieme jednoznačne určiť, do akého nového stavu sa dostaneme. Takto dostávame nový graf, ktorému zvykneme hovoriť „stavový priestor“. Vrcholy tohto grafu sú jednotlivé stavy, hrany zodpovedajú akciám, ktoré môžeme robiť.
Našu úlohu vieme teraz riešiť štandardným prehľadávaním. Keď chceme vedieť, či existuje postupnosť krokov, ktorá nás dostane do domu n, zaujíma nás vlastne otázka, či sa vieme dostať zo stavu „som pri dome 1, nebol som u babičky a mám 0 navštívených rodinných domov v rade“ do ľubovoľného stavu, v ktorom si pri dome n.
Prehľadávanie stavového priestoru (či už do šírky alebo hĺbky) má časovú zložitosť priamo úmernú jeho veľkosti, teda celkovému počtu jeho vrcholov a hrán. Už vieme, že vrcholov je $O(nk)$ – každý dom zodpovedá $O(k)$ stavom. V každom stave máme toľko možných akcií, koľko snehových tunelov vedie od dotyčného domu. A teda celkový počet akcií (hrán nášho nového grafu) je rádovo k-krát väčší ako celkový počet tunelov. Hrán teda máme $O(mk)$, čiže celková časová zložitosť je $O((m + n)k)$.

Pomocou tohto riešenia už vieme získať 5 bodov za súťažnú úlohu. Ak je v dedine nanajvýš 10 babičiek, je nanajvýš 10 možností, kde bude dedo Jozef. Pre každú možnosť si spravíme stavový graf pre všetky domy okrem toho práve zakázaného a prehľadávaním overíme, či existuje spôsob, ako sa dostať k domu n. Ak budú všetky tieto kontroly úspešné, dedina je priechodná, ak čo len jedna kontrola zlyhá, dedina priechodná nie je.

Vzorové riešenie 1

Zoberme stavový graf pre úplne celú dedinu a zistime, či sa vieme dostať do domu n. Ak nie, dedina určite nie je priechodná. Ak áno, sú dve možnosti. Ak sme sa pri nájdenom riešení nezastavili u žiadnej babičky, dedina priechodná zjavne je – toto riešenie môžeme použiť bez ohľadu na to, kde bude dedo Jozef.
Ostáva nám teda možnosť, že sme našli riešenie, pri ktorom sme sa zastavili u niektorej konkrétnej babičky. Prvé, čo teraz spravíme, je že zistíme, u ktorej to bolo. Toto vieme spraviť klasickou úpravou prehľadávania grafu, pri ktorej nie len zistíme, či je cieľ dosiahnuteľný, ale aj nájdeme jednu cestu doň. Na to si pre každý vrchol (stav) stačí pamätať, z ktorého vrcholu sme doň prvýkrát prišli. Akonáhle naše prehľadávanie úspešne príde do cieľa, môžeme sa po zapamätaných hranách postupne vrátiť na štart a takto zostrojiť cestu zo štartu do cieľa.
Keď sme toto do riešenia pridali, vieme teda, že sme našli riešenie, pri ktorom nás vykŕmi konkrétna babička B. Čo to znamená? Ak bude dedo Jozef u úplne hociktorej inej babičky, môžeme použiť toto riešenie. (Zjavne žiadny platný prechod dedinou nemôže navštíviť viac ako jednu babičku.) Jediné, čo potenciálne kazí priechodnosť dediny, je teda práve babička B. Ostáva nám teda práve jedna nezodpovedaná otázka: ak bude dedo Jozef práve u babičky B, vieme ho obísť?
No a toto vieme zistiť druhým prehľadávaním, pri ktorom do stavového priestoru zahrnieme všetky domy okrem domu babičky B.
Takto sme teda celú súťažnú úlohu vyriešili v čase $O((m + n)k)$, bez ohľadu na to, koľko existuje babičiek.

Vzorové riešenie 2

Existuje aj mnoho iných postupov, ako dosiahnuť rovnakú časovú zložitosť. Načrtneme ešte jeden, ktorý sa viac podobá na riešenia domáceho kola.

Najskôr zistíme, či existuje platná cesta do cieľa, ktorá obíde všetky babičky. Ak áno, môžeme ju vždy použiť a teda je dedina priechodná. Ak nie, každá cesta musí navštíviť práve jednu babičku.

V prvom rade si všimneme, že od babičky do cieľa musíme ísť už len cez prázdne domy. Vieme teda v pôvodnom grafe (vrcholy sú domy) spustiť prehľadávanie od domu n a zistiť, pre ktoré babičky toto vieme urobiť. Ostatné babičky od tejto chvíle môžeme úplne zabudnúť, ako keby neexistovali – zjavne sa u nich nikdy nemôžeme zastaviť, bez ohľadu na to, či tam je alebo nie je dedo Jozef. A naopak, akonáhle dosiahneme hociktorú z „dobrých“ babičiek, ktoré nám zostali, už je jasné, že sme vyhrali.

Teraz spustíme prehľadávanie celého stavového priestoru zo štandardného začiatku (sme hladní pri dome 1) bez nejakého špecifického cieľa – necháme ho prejsť celý stavový priestor a o úplne všetkých stavoch zistiť, či sú dosiahnuteľné.

No a už sme skoro hotoví. Ak sú dosiahnuteľné stavy kde sme u babičky pre aspoň dve rôzne *dobré* babičky, tak je dedina priechodná, ak je dosiahnuteľná nanajvýš jedna dobrá babička, tak dedina priechodná nie je.

Program uvedený nižšie implementuje toto riešenie.

Listing programu (C++)

```
#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

// Prehľadávanie do šírky od konca, šíriace sa len cez vrcholy typu 0
vector<bool> najdi_dosiahnutelne_vrcholy_od_konca() {
    vector<bool> dosiahnutelny(n, false);
    queue<int> q;

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

    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int w : e[v]) { // Iterujeme cez všetkých susedov
            if (dosiahnutelny[w]) continue;
            dosiahnutelny[w] = true; // Ako dosiahnuteľných označíme všetkých...
            if (typ[w] == 0) {
                q.push(w); // ... ale ďalej sa hýbeme len cez prázdne domy.
            }
        }
    }
    return dosiahnutelny;
}

// Prehľadávanie stavového grafu do šírky, začínajúc hladní v dome 0.
// Od babičiek ďalej nepokračuje.
// Vráti pre každý dom, či bol v nejakom stave dosiahnuteľný.
vector<bool> najdi_dosiahnutelne_vrcholy_zo_startu() {
    vector<vector<bool>> navstiveny;
    // Prvá súradnica je číslo domu, druhá dĺžka série rodinných domov
    navstiveny.resize(n);
    for (int i = 0; i < n; ++i)
        navstiveny[i].resize(k + 1, false);

    vector<bool> dosiahnutelny(n, false);
    queue<pair<int, int>> q;
    // Zložky záznamu vo fronte zodpovedajú súradniciam stavu

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

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

        for (int w : e[v.first]) {
            // Po k rodinných domoch musíme ísť ku prázdnemu
```

```
            if (v.second >= k && typ[w] != 0) continue;

            // Upravíme dĺžku série rodinných domov
            int nova_sytost = ((typ[w] == 1) ? v.second+1 : 0);

            // Ak sme už v tomto stave boli, ignorujeme ho
            if (navstiveny[w][nova_sytost]) continue;

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

    return dosiahnutelny;
}

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> dosiahnutelne_od_konca = najdi_dosiahnutelne_vrcholy_od_konca();
    vector<bool> dosiahnutelne_zo_startu = najdi_dosiahnutelne_vrcholy_zo_startu();

    // Ak existuje cesta bez návštevy babičky, vyhrali sme
    if (dosiahnutelne_zo_startu[n-1]) {
        cout << "ANO" << endl;
        return 0;
    }

    // Inak potrebujeme na výhru mať aspoň dve "dobré" babičky, ktoré sú dosiahnuteľné
    // aj od štartu ľubovoľne, aj z cieľa cez prázdne domy.
    int pocet_dobrych_babiciek = 0;
    for (int i = 0; i < n - 1; ++i) {
        if (typ[i] == 2 && dosiahnutelne_od_konca[i] && dosiahnutelne_zo_startu[i])
            ++pocet_dobrych_babiciek;
    }

    if (pocet_dobrych_babiciek >= 2)
        cout << "ANO" << endl;
    else
        cout << "NIE" << endl;
}
```

A-II-4 Externá pamäť

Podúloha A: selection sort

V podúlohe A malo riešenie dve hlavné súčasti: Najskôr bolo treba zvládnuť implementáciu tak, aby sme nemuseli kvôli každému prístupu do poľa pristupovať na disk, a následne správne odhadnúť časovú a komunikačnú zložitosť získanej implementácie.

Selection sort má síce už aj pri implementácii v pamäti kvadratickú časovú zložitosť, má však milú vlastnosť, ktorej zvykneme hovoriť *cache-friendly*: k prvkom v pamäti často pristupuje sekvenčne, vďaka čomu má potom nižšiu komunikačnú zložitosť ako niektoré iné kvadratické algoritmy. Toto má potom v praxi za dôsledok, že skutočná časová zložitosť tohto algoritmu má kvadratický člen prenásobený len malou konštantou. Moderné implementácie knižničných algoritmov pre usporiadanie poľa používajú nejaký takýto algoritmus pre úseky poľa po nejakú konštantnú dĺžku (napr. 64 prvkov), keďže vďaka malému konštantnému faktoru sú pre tieto vstupy rýchlejšie ako všeobecné algoritmy s lepšou asymptotickou časovou zložitosťou.

Nájdenie minima v úseku dĺžky d bude zjavne mať časovú zložitosť $\Theta(d)$. Tento úsek poľa je na disku uložený

v nanajvýš $\lfloor d/b \rfloor + 2$ rôznych blokoch. Keď si dáme pozor, aby sme každý blok len raz načítali do pamäte a potom sa pozreli na všetky relevantné prvky, dostaneme komunikačnú zložitosť $\Theta(d/b)$.
Následnú výmenu dvoch prvkov už implementujeme priamočiaro: načítame všetky bloky, ktorých sa výmena týka (niekedy je to jeden, niekedy sú dva), výmenu spravíme a bloky znova uložíme na disk. Toto má konštantnú časovú aj komunikačnú zložitosť, je to teda zanedbateľné v porovnaní s nájdením minima, ktoré výmene predchádzalo.
Celkovú časovú a komunikačnú zložitosť dostaneme ako súčet cez všetky možné hodnoty d. U časovej zložitosti zjavne dostaneme odhad $\Theta(n^2)$ – postupne prechádzame $n, n-1, n-2, \ldots$ prvkov. U komunikačnej zložitosti je to o čosi zložitejšie, ale nie príliš. Nie úplne exaktne môžeme uvažovať nasledovne: počas $\Theta(n^2)$ krokov algoritmu rádovo po každých b prístupoch k prvku príde jedno čítanie z disku. Čítaní z disku je preto $\Theta(n^2/b)$.
Exaktnejší výpočet je asi najjednoduchšie robiť od konca: Najskôr budeme mať niekoľko (nanajvýš b) iterácií, kedy pri hľadaní minima načítame do pamäte len jeden blok. Potom príde b iterácií kedy budeme prechádzať dva bloky, b iterácií s tromi blokmi, a tak ďalej až po iterácie počas ktorých budeme prechádzať rádovo n/b blokov. Máme teda n sčítancov s priemernou veľkosťou rádovo $n/2b$, čiže celková komunikačná zložitosť vychádza $\Theta(n^2/b)$.

Listing programu (Python)

```
# v pamäti už máme:
#     - konštantu B označujúcu veľkosť bloku na disku
#     - premennú N obsahujúcu dĺžku poľa

def vymen(i, j):
    # vymení prvky na indexoch i a j
    blok_i, blok_j = i//B, j//B
    if blok_i == blok_j:
        kus_pola = Read( blok_i )
        kus_pola[i%B], kus_pola[j%B] = kus_pola[j%B], kus_pola[i%B]
        Write( blok_i, kus_pola )
    else:
        kus_pola_i, kus_pola_j = Read( blok_i ), Read( blok_j )
        kus_pola_i[i%B], kus_pola_j[j%B] = kus_pola_j[j%B], kus_pola_i[i%B]
        Write( blok_i, kus_pola_i )
        Write( blok_j, kus_pola_j )

def najdi_index_minima(k):
    # nájde index najmenšieho prvku v x[k:N]

    # načítame do pamäte kus poľa obsahujúci prvok na indexe k
    # zapamätáme si jeho hodnotu
    kus_pola = Read( k//B )
    odpoved = k
    minimum = kus_pola[k%B]
    for i in range(k+1, N):
        if i%B == 0: # treba načítať nový blok do pamäte
            kus_pola = Read( i//B )
        if kus_pola[i%B] < minimum:
            odpoved, minimum = i, kus_pola[i%B]
    return odpoved

# selection sort:
for k in range(N):
    idx = najdi_index_minima(k)
    vymen( k, idx )
```

Pozor na rekurziu

Skôr, než sa pustíme do riešenia podúlohy B, ešte sa pristavíme pri rekurzii. Na problém s ňou sme už upozorňovali aj v študijnom texte. Každé volanie funkcie potrebuje svoj záznam na zásobníku (napr. aby sme vedeli, kde v programe pokračovať po ukončení volania funkcie) a tieto záznamy tiež spotrebúvajú pamäť, ktorú máme k dispozícii.
V bežnej praxi na problém nedostatku miesta na zásobníku narazíme len zriedkavo a aj to len u programov, ktoré používajú ozaj hlbokú rekurziu – napr. rekurzívne prehľadávanie obrovského grafu do hĺbky. A aj v týchto situáciách pretečenie zásobníka nastane len preto, že operačné systémy (a tiež niektoré interpretery, napr. Python) majú na veľkosť zásobníka vlastné umelé obmedzenie, ktoré je výrazne menšie ako celkové množstvo dostupnej pamäte.
U algoritmov, ktoré zásobník využívajú „len tak trošku“ nám ani nenapadne zamýšľať sa nad tým, že by nám

nemusel stačiť. Kanonickým príkladom algoritmu z tejto kategórie je triedenie mergesort. Ten vždy delí aktuálny kus poľa na polovicu, a teda počas celého behu triedenia na poli veľkosti n hĺbka rekurzie nikdy neprekročí $\log_2 n$: teda aj pre pole s miliardou prvkov budeme mať naraz na zásobníku nanajvýš 30 volaní našej rekurzívnej funkcie. No lenže v našom modeli sa hráme úplne inú hru. Zrazu máme pamäte len veľmi málo a nemáme nijaké obmedzenia na to, aké veľké môže byť n v porovnaní s veľkosťou bloku. A teda ani len algoritmus s hĺbkou rekurzie $\log_2 n$ si nemôžeme dovoliť – napr. pre vstupy, v ktorých je $n > 2^b$, by sme už potrebovali hĺbku zásobníka viac ako b.
Našťastie nás tento problém nijak výrazne neobmedzí – algoritmy pre triedenie, ktoré poznáme v rekurzívnej podobe, sa bez prílišnej námahy dajú upraviť aj do podoby, ktorá rekurziu nevyužíva.

Podúloha B: efektívne externé triedenie

Najjednoduchšie možnosti pre efektívne externé triedenie sú založené buď na úprave algoritmu quicksort alebo algoritmu mergesort. My si popíšeme mergesort, keďže u toho vieme ľahšie dosiahnuť, aby sme aj v najhoršom prípade mali časovú zložitosť $\Theta(n \log n)$.
Mergesort sa tradične zvykne implementovať ako rekurzívny algoritmus, rekurzia je ale len nástrojom pre naše pohodlie. Sústreďme sa len na samotný postup výroby usporiadaného poľa: na začiatku máme n jednoprvkových úsekov a postupne spájame kratšie usporiadané úseky do dlhších. Ak by sme zvládli rozumne implementovať takéto spájanie úsekov, mali by sme už z polovice vyhraté.
A skutočne, spájanie dvoch usporiadaných úsekov do jedného nevyzerá nijak komplikovane. Pri jeho klasickej implementácii vlastne len oba vstupné úseky postupne čítame zľava doprava a vždy nás zaujíma len prvý nespracovaný prvok z každého úseku. Práve spracúvané prvky sa nám do pamäte bez problémov zmestia a sekvenčné čítanie postupnosti vieme aj pri práci s diskom robiť efektívne.
Ale skôr, než sa pustíme do implementácie spájania úsekov, ešte spravíme jeden prípravný krok. Aby sme si zbytočne nekomplikovali život úsekmi, ktoré začínajú a končia kdesi uprostred našich blokov, nebudeme začínať od úsekov dĺžky 1. Namiesto toho jednoducho zoberieme zvlášť každý blok našej postupnosti do pamäte, tam ho celý usporiadame a uložíme naspäť na disk. Začínať náš vlastný mergesort teda budeme až v situácii, kde máme na disku n/b usporiadaných postupností, každá dĺžky b. Každý zo spájaných úsekov bude odteraz presne zaberať niekoľko celých blokov disku.
Pri spájaní dvoch usporiadaných úsekov do jedného použijeme voľné bloky za koncom postupnosti ako pomocnú pamäť. V pamäti si vždy budeme držať tri bloky: dva vstupné (z každej postupnosti ten obsahujúci aktuálny prvok) a jeden výstupný.

No a keď už máme túto pomocnú funkciu, budeme len iterovať spájanie dvojíc úsekov do jedného, až kým nedostaneme len jeden úsek. Jednotlivé fázy spájania budeme volať kolá. V každom kole si aktuálne úseky rozdelíme do dvojíc a každú dvojicu spojíme.
Ak napríklad máme 14 jednoblokových úsekov, tak v prvom kole postupne spojíme prvý s druhým, tretí so štvrtým, atď., čím dostaneme sedem dvojblokových úsekov. Z nich v druhom kole vzniknú už len štyri úseky (prvé tri budú štvorblokové, posledný bude kratší), v treťom kole spravíme zo štyroch úsekov dva a v poslednom kole spojíme tieto dva úseky a dostaneme usporiadané pole.

No a už nám ostáva len odhad zložitostí.
V prvom rade si rozmyslime, že naše vylepšenie (nezačínať od jednoprvkových úsekov ale každý blok samostatne usporiadať) na asymptotickú časovú zložitosť nemá vplyv. Počas usporadúvania blokov totiž spravíme rádovo rovnako veľa práce ako by sme spravili počas tých ušetrených kôl. (Ak by b bolo mocninou dvoch a my by sme použili na usporiadanie blokov mergesort, urobili by sme dokonca presne tie isté porovnania a spájania.)
No a v druhom rade si už len stačí uvedomiť, že iteratívne implementovaný mergesort má rovnakú asymptotickú časovú zložitosť ako ten klasický rekurzívny: prebehne postupne $O(\log n)$ kôl a v každom spravíme $O(n)$ operácií. Naša implementácia má preto časovú zložitosť $O(n \log n)$.
Ako je to s komunikačnou zložitosťou? Pri úvodnom triedení blokov spravíme n/b čítaní a n/b zápisov. Keď spájame dokopy dva úseky, ktoré majú dokopy k blokov, spravíme $2k$ čítaní a $2k$ zápisov blokov. Dokopy teda v každom kole spravíme nanajvýš $4(n/b)$ operácií s diskom. No a kôl má naša implementácia $\lceil \log_2(n/b) \rceil$ kôl, takže dokopy spravíme nanajvýš $2(n/b) + 4(n/b)\lceil \log_2(n/b) \rceil$ operácií s diskom.

Listing programu (Python)

```
from model import read_input_sequence
read_input_sequence('4a.in')
from model import Read, Write, B, N

# spoj usporiadané postupnosti tvorené blokmi s číslami [z1:z2] a [z2:z3]
def spoj_useky(z1, z2, z3):
    i, j = z1*B, z2*B
    vystup = []
    zout = N // B
    while i < z2*B or j < z3*B:
        # ak ideme spracovať prvý prvok bloku, musíme si ho najskôr načítať z disku
        if i < z2*B and i%B == 0: vstup1 = Read(i//B)
        if j < z3*B and j%B == 0: vstup2 = Read(j//B)

        # ak je niektorá postupnosť prázdna, zoberieme z druhej, inak zoberieme menší
        if (j == z3*B) or (i < z2*B and vstup1[i%B] < vstup2[j%B]):
            vystup.append( vstup1[i%B] )
            i += 1
        else:
            vystup.append( vstup2[j%B] )
            j += 1

        # ak už máme plný výstupný blok, uložíme ho na disk
        if len(vystup) == B:
            Write(zout, vystup)
            vystup = []
            zout += 1

    # a už len presunieme spojenú postupnosť z pomocnej "pamäte" naspäť
    for i in range(z3-z1):
        Write( z1+i, Read( N//B + i ) )

# mergesort

# usporiadame jednotlivé bloky
for i in range(N//B):
    Write( i, sorted( Read(i) ) )

# kým máme viac ako jeden úsek, robíme kolo spájania
pocet_usekov, dlzka_useku = N//B, 1

while pocet_usekov > 1:
    for i in range(pocet_usekov//2):
        spoj_useky( (2*i)*dlzka_useku, (2*i+1)*dlzka_useku, min(N//B, (2*i+2)*dlzka_useku) )
    pocet_usekov = (pocet_usekov + 1) // 2
    dlzka_useku *= 2
```

Podúloha C

Hlavná vec, ktorá sa zmení, keď máme k dispozícii viac pamäte, je, že si môžeme dovoliť spájať viac postupností naraz.

V prvom rade sa zamyslime, ako presne to robiť – presnejšie, že to nemôžeme robiť úplne hlúpo. Ak by sme pri spájaní p postupností do jednej postupovali tak, že v každom kole sa pozrieme na všetkých p začiatkov a vyberieme najmenší z nich, dostali by sme pre n-prvkové postupnosti časovú zložitosť $O(np)$. Klasický mergesort však vie tieto postupnosti pospájať v čase $O(n \log p)$: bude mať $O(\log p)$ kôl spájania a každé z nich bude mať časovú zložitosť $O(n)$.

Našťastie takúto istú časovú zložitosť vieme dosiahnuť aj my jednoduchým vylepšením. Na začiatku zoberieme z každej postupnosti jej prvý (najmenší) prvok a všetky tieto prvky vložíme do prioritnej fronty. (Tú môžeme implementovať napr. ako binárnu haldu s minimom v koreni.) No a potom spájanie postupností prebieha tak, že vždy z prioritnej fronty vyberieme minimum, to pridáme do usporiadanej postupnosti, a následne do prioritnej fronty pridáme ďalší prvok z tej postupnosti, z ktorej bol práve spracovaný. Takto v každom kole potrebujeme len čas $O(\log p)$ na nájdenie a spracovanie aktuálneho minima, a teda p postupností dĺžky n spojíme v rovnakej asymptotickej časovej zložitosti ako klasický mergesort.

No dobre, ale ak tým nezmeníme asymptotickú časovú zložitosť, prečo to vôbec robiť? Práve preto, že tým vieme zlepšiť komunikačnú zložitosť (a tak často v praxi aj reálny čas behu programu).

Spočítajme si, čo sa stane, keď budeme pri mergesorte vždy spájať dokopy až p postupností naraz. Zmení sa nám počet kôl spájania: namiesto $\log_2 n$ ich bude $\log_p n$. No a naďalej bude platiť, že v každom kole každý blok vstupu len koštantne veľakrát načítame a zapíšeme, takže komunikačná zložitosť bude $O((n/b) \log_p n)$.

Zjavne čím väčšie p vieme zvoliť, tým väčší je základ logaritmu a teda tým menšia je výsledná komunikačná zložitosť. Aké najväčšie môžeme zvoliť p? V pamäti naraz potrebujeme mať $p+1$ blokov (po jednom vstupnom

pre každú postupnosť a jeden výstupný) a prioritnú frontu (tá má len prvok na postupnosť, čiže zaberá zhruba b-krát menej miesta). Ak napríklad zvolíme $p = m/(2b)$, zaberieme o čosi viac ako polovicu dostupnej pamäte, a to je tak akurát. Dostávame teda výslednú komunikačnú zložitosť $O((n/b) \log_{m/(2b)} n)$.
Na záver ešte uvedieme, že sa dá dokázať, že tento algoritmus má pre naše zadanie asymptoticky optimálnu časovú aj komunikačnú zložitosť.

TRIDSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2021